

普通高等教育“十一五”国家级规划教材



21世纪大学本科计算机专业系列教材

袁春风 编著

计算机组成与系统结构

http://www.tup.com.cn

- 根据教育部“高等学校计算机科学与技术专业规范”组织编写
- 与美国 ACM 和 IEEE CS *Computing Curricula* 最新进展同步
- 教育部–微软精品课程教材
- 远程教育国家精品课程教材

清华大学出版社

普通高等教育“十一五”国家级规划教材

21世纪大学本科计算机专业系列教材

计算机组成与系统结构

袁春风　编著

黄宜华　审校

清华大学出版社

北京

内 容 简 介

本书主要介绍计算机组成与系统结构涉及的相关概念、理论和技术内容，主要包括指令集体系结构、数据的表示和存储，以及实现指令集体系结构的计算机各部件的内部工作原理、组成结构及其相互连接关系。本书共分9章：第1章对计算机系统及其性能评价进行概述性介绍；第2～3章主要介绍数据的机器级表示、运算，以及运算部件的结构与设计；第4章主要介绍包含主存、cache和虚拟存储器在内的存储器分层体系结构；第5～7章介绍指令系统以及各种CPU设计技术；第8～9章介绍总线互连以及输入输出系统。

本书内容详尽、反映现实、概念清楚、通俗易懂、实例丰富，并提供大量典型习题以供读者练习。本书可以作为计算机专业本科或大专院校学生计算机组成原理与系统结构课程的教材，也可以作为有关专业研究生或计算机技术人员的参考书。

图书在版编目（CIP）数据

计算机组成与系统结构/袁春风编著．--北京：清华大学出版社，2010.4
（21世纪大学本科计算机专业系列教材）
ISBN 978-7-302-21905-7

Ⅰ．①计… Ⅱ．①袁… Ⅲ．①计算机组织－高等学校－教材②计算机体系结构－高等学校－教材 Ⅳ．①TP302.1②TP303

中国版本图书馆CIP数据核字(2010)第013214号

责任编辑：张瑞庆 薛 阳
责任校对：焦丽丽
责任印制：

出版发行：清华大学出版社 **地 址**：北京清华大学学研大厦A座
http://www.tup.com.cn **邮 编**：100084
社 总 机：010-62770175 **邮 购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经 销：全国新华书店
开 本：185×260 **印 张**：27.75 **字 数**：676千字
版 次：2010年4月第1版 **印 次**：2010年4月第1次印刷
印 数：1～ 000
定 价：39.00元

产品编号：-01

前　言

FOREWORD

计算机组成(computer organization)是指计算机主要功能部件的组成结构、逻辑设计及功能部件间的相互连接关系。计算机系统结构(computer architecture)的经典定义是指程序设计者(主要指低级语言程序员或编译程序设计者)所看到的计算机系统的属性,即计算机的功能特性和概念性结构,也称指令集体系结构(Instruction Set Architecture,ISA),包括:数据类型及数据格式,指令格式,寻址方式和可访问空间大小,程序可访问的寄存器个数、位数和编号,控制寄存器的定义,I/O空间的编址方式,中断结构,机器工作状态的定义和切换,输入输出数据传送方式,存储保护方式等。

本书主要介绍单处理器计算机系统的组成与系统结构涉及的相关内容。在计算机系统层次结构中,这些内容位于软件和硬件的结合处,不仅涉及计算机硬件设计和指令系统设计,还涉及操作系统、编译程序和程序设计等部分软件设计技术,是整个计算机系统中最核心的部分。

1. 本书的写作思路和内容组织

计算机组成与系统结构这两部分涉及的内容相互融合,密不可分。无论是国内还是国外,很多高校都逐渐把计算机组成原理和系统结构课程的内容有机结合起来;甚至国外一些经典教材还把密切相关的软件设计的内容也融合到一起。这种方式可以加深读者对计算机软硬件系统的整体化理解,并有效地增强对学生的计算机系统设计能力的培养。

本书在总结和借鉴国外著名高校使用的教材、教案、教学理念和教学方法的基础上,力图以"培养学生现代计算机系统设计能力"为目标,贯彻"从程序设计视角出发、强调软硬件关联与协同、以CPU设计为核心"的组织思路,试图改变国内同类教材通常的就硬件讲硬件、软硬件分离的传统内容组织方式,以系统化观点全面地介绍计算机组成和系统结构的相关知识和技术。

为了体现以上思路和目标,本书在以下几个方面进行了重点考虑和内容组织:

(1) 首先基于"高级语言程序→汇编语言程序→机器指令序列→控制信号"的路线,展现程序从编程设计、转换翻译到最终在CPU上运行的整个过程;在此基础上,用计算机系统层次化的观点阐述计算机组成与系统结构课程在整个计算机系统中的位置、内容和作用,从而为清晰了解本课程的内容和作用、为全面建立计算机软硬件系统的整体概念打下基础。

(2) 将指令执行过程和异常、中断、存储访问、I/O访问等重要概念和技术结合起来进行介绍,力求清晰说明CPU执行指令过程中硬件与操作系统相互切换和协同工作的处理过程,使读者深刻理解软硬件系统之间的关系与协同工作过程。

(3) 在讲述与程序设计有密切关系的体系结构内容(如数据表示、信息存放、操作数寻址、过程调用、程序访问局部性等)时,试图通过对硬件设计与程序设计的关系的说明,使读者建立“从程序员视角理解计算机硬件系统设计,从硬件设计的视角理解程序设计与执行”的思想,力图在提高读者硬件设计能力的同时,也增强其进行高效的和系统化的程序设计的能力。

(4) 在国内“计算机组成原理”教材传统内容基础上增加指令流水线设计的详细内容,依照“最简单的 IAS 计算机 CPU→总线式 CPU→单周期 CPU→多周期 CPU→基本流水线 CPU→动态超标量超流水线 CPU”的次序,循序渐进地介绍 CPU 设计技术及其发展过程,以 MIPS 处理器和 Pentium 4 处理器为蓝本,力图使读者全面深入地掌握现代计算机的 CPU 设计技术。

(5) 结合指令流水线技术介绍基于流水线的编译优化技术,使读者对编译技术与指令流水线实现技术之间的密切关系有一定的认识和理解。

2. 本书各章节的主要内容

本书共有 9 章,各章主要内容如下:

第 1 章(计算机系统概述)主要介绍冯·诺依曼结构的特点、计算机硬件的基本组成、计算机软件设计和执行过程、计算机系统层次结构,以及系统性能评价方法;

第 2 章(数据的机器级表示)主要介绍无符号数和有符号整数的表示、IEEE 754 浮点数标准、西文字符和汉字的编码表示、大端/小端存放顺序及对齐方式,以及常用检/纠错码表示与使用方法等;

第 3 章(运算方法和运算部件)主要介绍各类定点数和浮点数的运算方法和相应的运算部件,以及核心运算部件 ALU 的功能和设计实现等;

第 4 章(存储器分层体系结构)主要介绍存储器分层结构的概念、半导体存储器的组织、多模块存储器、cache 的基本原理、cache 和主存间的映射关系及替换算法、虚拟存储器的基本概念、页表结构、缺页异常、TLB 的实现技术;

第 5 章(指令系统)主要介绍高级语言与低级语言的关系、指令格式、操作数类型、寻址方式、操作类型、硬件对过程的支持、用户程序在虚存空间的配置和划分等技术;

第 6 章(中央处理器)主要介绍 CPU 的基本功能和内部结构、指令执行过程、数据通路的基本组成和定时、单周期和多周期数据通路、硬连线路和微程序控制器、异常和中断等概念和技术;

第 7 章(指令流水线)主要介绍指令流水线的基本原理、流水段寄存器的概念、流水线数据通路的设计、流水线的控制信号、结构冒险及其处理、数据冒险及其处理、转发技术、控制冒险及其处理、分支预测原理、超标量和动态流水线的概念和技术;

第 8 章(系统总线)主要介绍总线基本概念、总线裁决、总线定时、总线标准及其现代计算机内部的总线互连结构;

第 9 章(输入输出组织)主要介绍常用输入输出外设和磁盘存储器的工作原理、I/O 接口的结构、I/O 端口编址方式、程序查询 I/O 方式、中断 I/O 方式和 DMA 方式等。

3. 关于本书使用的一些建议

本书可作为“计算机组成原理”课程的教材,也可作为“计算机组成原理实验”课程和“计算机系统结构”课程的教学参考书,特别是对于不专门开设“计算机系统结构”课程的院校,

使用本书作为“计算机组成与系统结构”课程的教材是比较合适的。

本书力求用历史的、系统的观点全面深入地介绍计算机组成与系统结构所涉及的重要概念和知识体系，并力求准确、清晰地阐述相关内容之间的关联，因而，内容较多、篇幅较大，为此，本书对课堂教学内容和课后阅读内容进行了区分，在目录和正文的章节标题处加 * 标注表示可作为课后阅读的内容。本书作为教材使用时，可以根据不同的教学目标和课时限制，有选择地进行内容裁剪和选择。

对于本书的使用，具体建议如下：

（1）课堂教学应以主干内容为主，力求完整给出知识框架体系，并着重讲清楚相关概念之间的联系。

（2）标注为 * 的内容是可以跳过而不影响阅读连贯性的部分，主要有以下三类：简单易懂的基础性内容、具体实现方面的细节内容和在技术层面上更加深入的内容。这些内容对深入理解课程的整体核心内容是非常有帮助的。因此，在课时允许的情况下，可以选择其中的一部分进行课堂讲解；在课时不允许的情况下，也尽量安排学生进行课后阅读。

（3）书中每个重要的知识点和概念后面都有一些例子，可选择部分重要的、难懂的例子在课堂上讲解，而大部分可留给学生自学。

（4）习题中列出的概念术语基本涵盖了相应章节的主要概念，可以让学生对照检查是否全部清楚其含义；习题中列出的简答问题是相应章节重要的基本问题，可以通过对照检查以判断学生对相应章节内容的掌握程度；对于综合运用题，如果与程序设计相关，则可用编程方式来求解或验证，这样做，对学生深刻理解课程内容有帮助。

（5）本书在 CPU 设计方面给出了比较具体的实现方案，相关内容可以作为基于 FPGA 和硬件描述语言进行 CPU 设计实验的参考资料。

（6）书后给出了部分国际一流大学的相关课程网站网址，可以到这些网站找到课堂讲义、习题参考答案，以及更多的相关教辅资料。

4. 致谢

在本书的编写过程中，得到了张福炎教授的悉心指导；黄宜华教授从书稿的篇章结构到内容各方面都提出了许多宝贵的意见，进行了修改，并对全书内容进行了全面细致的审核和校对；书中有关 CPU 设计的最初图稿和内容组织思路由陈贵海教授提供；此外，武港山教授、俞建新、吴海军、张泽生、蔡晓燕等老师也对本书提出了许多宝贵的意见；杨晓亮、肖韬、翁基伟、刘长辉、宗恒、莫志刚、叶俊杰等研究生对相关章节的内容和习题分别进行了校对和试做，并提出了许多宝贵的意见和修改建议。在此对以上各位老师和研究生一并表示衷心的感谢。

本书是作者在南京大学从事“计算机组成与系统结构”课程教学近 20 年来所积累的讲稿内容的基础上编写而成的，感谢各位同仁和各届学生对讲稿内容所提出的宝贵的反馈和改进意见，使得本教材的内容得以不断地改进和完善。

5. 结束语

本书广泛参考了国内外相关的经典教材和教案，在内容上力求做到取材先进并反映技术发展现状；在内容的组织和描述上力求概念准确、语言通俗易懂、实例深入浅出，并尽量利用图示和实例来解释和说明问题。相信只要读者具有数字逻辑电路和程序设计的一些基本

概念，就能通过使用本书较为全面地理解相关的基础理论、技术和知识体系；理解和掌握现代计算机的指令集体系结构；理解和掌握ALU、乘法器、存储器、I/O接口、总线等各种基本部件以及流水线CPU的基本工作原理、结构和设计技术。

但是，由于计算机组成与系统结构相关的基础理论和技术在不断发展，新的思想、概念、技术和方法不断涌现，加之作者水平有限，在编写中难免存在不当或遗漏之处，恳请广大读者对本书的不足之处给予指正，以便在后续的版本中予以改进。

为了更好地提供教学支持，并逐步形成一个广泛、方便、有效的课程教学交流空间，我们构建了相应的课程网站。其中包括每个章节的主要内容提要、学习目标和要求、课堂讲义、作业及参考答案、概念术语、常见问题、课后练习、例题分析等，此外还包括课程教学大纲、课程实验内容、虚拟实验、装机实践、课程动画演示和计算机小百科等。我们还将不断更新课程内容，并尽快将课程教学的视频文件放到课程网站上，以供大家参考。

课程网址如下（如果只是浏览课程内容，建议使用后面两个网址，速度较快）：

http://graphics.nju.edu.cn/（用户名和密码都是student，可进入讨论区进行交流）。

http://media.njude.com.cn/course/jsjzcyl/index.htm。

http://tres.njude.com.cn/msmk/jsjzcyl/index.htm。

作　者

2010年2月

目　录

CONTENTS

第 1 章　计算机系统概述 …… 1
1.1　计算机的功能和特性 …… 1
1.2　计算机的发展历程 …… 2
*1.2.1　电子计算机的诞生 …… 2
*1.2.2　第一代计算机 …… 2
*1.2.3　第二代计算机 …… 3
*1.2.4　第三代计算机 …… 4
*1.2.5　第四代计算机 …… 4
1.3　计算机系统的组成 …… 6
1.3.1　计算机硬件 …… 6
1.3.2　计算机软件 …… 9
1.4　计算机系统的层次化结构 …… 10
1.4.1　最终用户眼中的计算机 …… 10
1.4.2　系统管理员眼中的计算机 …… 10
1.4.3　应用程序员眼中的计算机 …… 11
1.4.4　系统程序员眼中的计算机 …… 11
1.4.5　程序开发与执行过程 …… 11
1.5　本教材的主要内容和组织结构 …… 14
1.6　计算机系统性能评价 …… 16
1.6.1　计算机性能的定义 …… 16
1.6.2　计算机性能的测试 …… 16
1.6.3　用指令执行速度进行性能评估 …… 19
1.6.4　用基准程序进行性能评估 …… 20
1.7　本章小结 …… 21
习题 1 …… 22
第 2 章　数据的机器级表示 …… 25
2.1　数制和编码 …… 25
*2.1.1　信息的二进制编码 …… 25

*2.1.2 进位计数制 …… 27
2.1.3 定点与浮点表示 …… 31
2.1.4 定点数的编码表示 …… 31
2.2 整数的表示 …… 36
2.2.1 无符号整数的表示 …… 36
2.2.2 带符号整数的表示 …… 37
*2.2.3 C语言中的整数类型 …… 37
2.3 实数的表示 …… 38
2.3.1 浮点数的表示格式 …… 38
2.3.2 浮点数的规格化 …… 40
2.3.3 IEEE 754浮点数标准 …… 41
*2.3.4 C语言中的浮点数类型 …… 45
2.4 十进制数的表示 …… 47
*2.4.1 用ASCII码字符表示 …… 47
2.4.2 用BCD码表示 …… 48
2.5 非数值数据的编码表示 …… 49
2.5.1 逻辑值 …… 49
2.5.2 西文字符 …… 50
*2.5.3 汉字字符 …… 51
2.6 数据的宽度和存储 …… 53
2.6.1 数据的宽度和单位 …… 53
2.6.2 数据的存储和排列顺序 …… 55
2.7 数据校验码 …… 58
2.7.1 奇偶校验码 …… 60
2.7.2 海明校验码 …… 60
*2.7.3 循环冗余校验码 …… 64
2.8 本章小结 …… 67
习题2 …… 68

第3章 运算方法和运算部件 …… 72

3.1 高级语言和机器指令中的运算 …… 72
*3.1.1 C程序中涉及的运算 …… 72
*3.1.2 MIPS指令中涉及的运算 …… 75
3.2 基本运算部件 …… 77
3.2.1 串行进位加法器 …… 78
*3.2.2 进位选择加法器 …… 79
3.2.3 并行进位加法器 …… 79
3.2.4 算术逻辑部件 …… 82
3.3 定点数运算 …… 85

3.3.1　补码加减运算 …… 86
*3.3.2　原码加减运算 …… 88
*3.3.3　移码加减运算 …… 89
3.3.4　原码乘法运算 …… 90
3.3.5　补码乘法运算 …… 95
*3.3.6　快速乘法器 …… 99
3.3.7　原码除法运算 …… 101
*3.3.8　补码除法运算 …… 108
*3.3.9　阵列除法器 …… 112
3.4　浮点数运算 …… 113
3.4.1　浮点数加减运算 …… 113
*3.4.2　浮点数乘除运算 …… 117
3.5　运算部件的组成 …… 120
*3.5.1　定点运算部件 …… 120
*3.5.2　浮点运算部件 …… 122
3.6　十进制数加减运算 …… 124
3.7　本章小结 …… 125
习题 3 …… 126

第 4 章　存储器分层体系结构 …… 130

4.1　存储器概述 …… 130
4.1.1　存储器的分类 …… 130
4.1.2　主存储器的组成和基本操作 …… 131
4.1.3　存储器的主要性能指标 …… 132
4.1.4　存储器的层次化结构 …… 133
4.2　半导体随机存取存储器 …… 134
4.2.1　基本存储元件 …… 134
4.2.2　静态 RAM 芯片 …… 135
4.2.3　动态 RAM 芯片 …… 138
4.3　半导体只读存储器和 Flash 存储器 …… 141
4.3.1　半导体只读存储器 …… 141
4.3.2　半导体 Flash 存储器 …… 142
4.4　存储器芯片的扩展及其与 CPU 的连接 …… 143
4.4.1　存储器芯片的扩展 …… 143
4.4.2　存储器芯片与 CPU 的连接 …… 144
4.5　并行存储器结构技术 …… 146
4.5.1　双口存储器 …… 146
4.5.2　多模块存储器 …… 147
4.6　高速缓冲存储器 …… 149

4.6.1 程序访问的局部性…… 149
4.6.2 cache 的基本工作原理 …… 151
4.6.3 cache 行和主存块之间的映射方式 …… 153
4.6.4 cache 中主存块的替换算法 …… 160
4.6.5 cache 的一致性问题 …… 164
4.6.6 cache 性能评估 …… 165
*4.6.7 影响 cache 性能的因素 …… 166
*4.6.8 cache 结构举例 …… 169
4.7 虚拟存储器 …… 170
*4.7.1 进程与进程的上下文切换 …… 171
*4.7.2 存储器管理 …… 172
4.7.3 虚拟地址空间…… 174
4.7.4 虚拟存储器的实现…… 176
*4.7.5 存储保护 …… 183
4.8 本章小结 …… 184
习题 4 …… 186

第 5 章 指令系统 …… 192

5.1 指令格式设计 …… 192
5.1.1 指令地址码的个数…… 192
5.1.2 指令格式设计原则…… 193
5.2 指令系统设计 …… 194
5.2.1 基本设计问题…… 194
5.2.2 操作数类型…… 195
5.2.3 寻址方式…… 195
5.2.4 操作类型…… 199
5.2.5 操作码编码…… 200
*5.2.6 条件码的生成与使用 …… 202
5.2.7 指令系统设计风格…… 203
5.3 指令系统实例 …… 205
*5.3.1 Pentium 指令系统 …… 205
*5.3.2 Power PC 指令系统 …… 208
*5.3.3 MMX 和 SIMD 指令技术 …… 209
5.4 程序的机器级表示 …… 210
*5.4.1 MIPS 汇编语言和机器语言 …… 210
*5.4.2 选择结构的机器代码表示 …… 214
*5.4.3 循环结构的机器代码表示 …… 215
*5.4.4 过程调用的机器代码表示 …… 216
5.5 本章小结 …… 223

习题 5 ······ 225

第 6 章　中央处理器 ······ 229

6.1　CPU 概述 ······ 229
6.1.1　指令执行过程 ······ 229
6.1.2　CPU 的基本功能 ······ 230
6.1.3　CPU 的基本组成 ······ 231
6.1.4　数据通路的基本结构 ······ 232
6.2　单周期处理器设计 ······ 240
6.2.1　指令功能的描述 ······ 241
6.2.2　数据通路的设计 ······ 242
6.2.3　控制逻辑单元的设计 ······ 251
6.2.4　时钟周期的确定 ······ 258
6.3　多周期处理器设计 ······ 259
*6.3.1　信号竞争问题 ······ 259
*6.3.2　指令执行状态分析 ······ 260
*6.3.3　硬连线路控制器设计 ······ 263
6.4　微程序控制器设计 ······ 265
*6.4.1　Wilkes 微程序控制器 ······ 266
6.4.2　微程序控制器的结构 ······ 267
6.4.3　微命令编码和微指令格式 ······ 268
6.4.4　微指令地址的确定 ······ 273
6.5　异常和中断处理 ······ 276
6.5.1　基本概念 ······ 276
6.5.2　异常处理过程 ······ 278
*6.5.3　带异常处理的处理器设计 ······ 279
6.6　本章小结 ······ 282
习题 6 ······ 284

第 7 章　指令流水线 ······ 287

7.1　流水线概述 ······ 287
7.1.1　流水线的执行效率 ······ 287
7.1.2　适合流水线的指令集特征 ······ 288
7.2　流水线处理器的实现 ······ 289
7.2.1　每条指令的流水段分析 ······ 289
7.2.2　流水线数据通路的设计 ······ 290
7.2.3　流水线控制器的设计 ······ 295
7.3　流水线冒险及其处理 ······ 296
7.3.1　结构冒险 ······ 296

7.3.2 数据冒险 …… 297
7.3.3 控制冒险 …… 304
*7.3.4 访问缺失引起的流水线阻塞 …… 310
7.4 高级流水线技术 …… 312
*7.4.1 静态多发射处理器 …… 313
*7.4.2 动态多发射处理器 …… 317
*7.4.3 Pentium 4 处理器的流水线结构 …… 321
7.5 本章小结 …… 324
习题 7 …… 326

第 8 章 系统总线 …… 329

8.1 总线的基本概念 …… 329
*8.1.1 总线的特性和分类 …… 329
8.1.2 系统总线的组成 …… 330
8.2 总线设计的要素 …… 331
8.2.1 信号线类型 …… 331
8.2.2 总线事务类型 …… 332
8.2.3 总线带宽 …… 333
8.2.4 总线裁决 …… 333
8.2.5 定时方式 …… 338
*8.3 总线接口单元 …… 343
8.4 总线标准 …… 344
*8.4.1 ISA 总线 …… 344
*8.4.2 EISA 总线 …… 345
*8.4.3 PCI 总线 …… 345
8.5 总线结构 …… 351
*8.5.1 单总线结构 …… 351
*8.5.2 双总线结构 …… 351
8.5.3 多总线结构 …… 352
8.6 本章小结 …… 354
习题 8 …… 355

第 9 章 输入输出组织 …… 358

9.1 外部设备的分类与特点 …… 358
9.1.1 外设的分类 …… 358
9.1.2 外设的特点 …… 359
9.2 输入设备和输出设备 …… 359
*9.2.1 键盘 …… 359
*9.2.2 鼠标器 …… 361

*9.2.3　打印机 …… 361
*9.2.4　显示器 …… 364
9.3　外部存储设备 …… 366
9.3.1　磁表面存储原理 …… 366
9.3.2　硬盘存储器 …… 370
*9.3.3　磁带存储器 …… 377
*9.3.4　光盘存储器 …… 378
9.4　I/O 接口 …… 380
9.4.1　I/O 接口的功能 …… 380
9.4.2　I/O 接口的通用结构 …… 381
*9.4.3　操作系统对 I/O 的支持 …… 382
9.4.4　I/O 端口及其编址 …… 384
*9.4.5　I/O 接口的分类 …… 386
9.4.6　并行传输和串行传输 …… 387
*9.4.7　I/O 接口举例 …… 390
9.5　I/O 数据传送控制方式 …… 394
9.5.1　程序直接控制 I/O 方式 …… 395
9.5.2　程序中断 I/O 方式 …… 398
9.5.3　DMA 方式 …… 407
*9.5.4　通道和 I/O 处理器方式 …… 413
9.6　本章小结 …… 416
习题 9 …… 418

参考文献 …… 422

第 1 章 计算机系统概述

本章主要介绍计算机系统的基本功能、发展历程、计算机系统的组成、计算机系统层次化结构以及计算机系统的性能评价。

1.1 计算机的功能和特性

计算机是一种能自动对数字化信息进行算术和逻辑运算的高速处理装置。也就是说，计算机处理的对象是数字化信息，处理的手段是算术和逻辑运算，处理的方式是自动的。计算机不仅具有数据处理功能，还具有数据存储、数据传送等功能，因此，计算机与算盘以及各类机械式计算器有本质的差别。

数据处理是计算机系统最基本的功能，计算机不仅可以进行加、减、乘、除等基本算术运算，也可以进行与、或、非等逻辑运算；处理的数据不仅可以是日常生活中使用的十进制数据，也可以是文字、图形、图像、声音、视频等非数值化的各种多媒体信息。

数据存储功能是计算机能采用自动工作方式的最基本保证。计算机中提供的存储器使得程序和数据能事先被存储，并在需要时被取出自动执行。计算机中有各类存储部件，大量的文件信息需要长期存储在计算机系统中，因此有能够长期保存信息的像磁盘存储器那样的非易失性存储器；正在执行的程序和处理的数据需要存放在快速存储器中，因此有半导体元器件构成的随机访问存储器等。

数据传送功能是指计算机内部的各个功能部件之间、计算机主机与外部设备之间、各个计算机系统之间进行信息交换的操作功能。例如，进行数据处理的部件需要从数据存储部件中读取信息或写入信息；输入设备的信息需要送到存储设备保存或送到数据处理部件进行计算；一台计算机产生的数据需要送到另一台计算机，因此，计算机系统中不可避免地需要进行数据传送。

数据处理、数据存储和数据传送的功能最终是通过执行指令来完成的，而计算机指令的执行过程由控制器产生的控制信号来控制。

对照上述基本功能，计算机中需要有对数据进行处理、存储和传送的基本功能部件，以及控制这些功能部件操作的控制部件。通常把进行数据处理的部件称为运算部件，主要运算部件是算术逻辑运算部件；把进行数据存储的部件称为存储器，主要分外存（storage）和

内存(memory);把进行数据传送的部件称为互连部件,主要有总线(bus)、桥接器和 I/O 接口等。

计算机具有高速、通用、准确和智能等特性。计算机的主要核心部件采用高速电子元器件制造,这为计算机快速处理提供了基本保证。通用性体现在两个方面:一是它所处理信息的多样化,可以是各种数值信息和非数值信息;二是计算机应用极其广泛,只要现实世界中某个问题能找到相应的算法,就能编制成程序通过计算机执行来加以解决。此外,计算机强大的计算和自动逻辑推理能力为计算机的准确性和智能化提供了重要基础。

1.2 计算机的发展历程

*1.2.1 电子计算机的诞生

世界上第一台电子计算机是 1946 年在美国诞生的 ENIAC,其设计师是美国宾夕法尼亚大学的莫齐利(Mauchly)和他的学生艾克特(Eckert)。莫齐利于 1932 年获得著名的霍普金斯大学物理学博士学位并留校任教,1941 年转入宾夕法尼亚大学,他常常为物理学研究中屡屡出现的大量枯燥、繁琐的数学计算而头痛,渴望电子计算机帮忙。一天,他偶然发现爱荷华州立大学的阿塔纳索夫教授正在试制电子计算机,莫齐利深感鼓舞,立即启程拜访。阿塔纳索夫教授热情地接待了这位志同道合的伙伴,毫无保留地介绍了研制情况,并无私地把有关电子计算机设计的珍贵笔记本郑重地交给了莫齐利。莫齐利认真研究了阿塔纳索夫的方案,凭着他特有的聪明才智,加上坚实的数学和物理基础以及电子学方面的丰富实践经验,于 1942 年写出了一份题为《高速电子管装置的使用》的报告。该报告很快引起了一个年轻人——23 岁的研究生艾克特——的兴趣,于是,师生密切协作,开始了计算机的研制。当时正值第二次世界大战期间,军方急需一种高速电子装置来解决弹道的复杂计算问题,莫齐利与艾克特的方案在 1943 年得到了军方的支持。在冯·诺依曼等人的帮助下,他们经过两年多的努力,终于研制成了第一台电子计算机。1946 年 2 月,美国陆军军械部与摩尔学院共同举行新闻发布会,宣布了第一台电子计算机 ENIAC(Electronic Numerical Integrator and Computer,电子数字积分机和计算机)研制成功的消息。

ENIAC 能进行每秒 5000 次加法运算,每秒 50 次乘法运算以及平方和立方、sin 和 cos 等函数数值运算。当时主要用它来进行弹道参数计算,60 秒钟射程的弹道计算时间由原来的 20 分钟一下子缩短到仅需 30 秒,ENIAC 的名声不胫而走。它是个庞然大物,耗资 40 万美元,使用了 18 000 个真空管,重 30 吨,占地面积 170 平方米,耗电 160 千瓦,第一次开机时甚至整个费城地区的照明灯都闪烁变暗。该机正式运行到 1955 年 10 月 2 日,这 10 年间共运行了 80 223 个小时。

自从第一台电子计算机 ENIAC 诞生后,人类社会进入了一个崭新的电子计算和信息化时代。计算机硬件早期的发展受电子开关器件的影响极大,为此,传统上人们以元器件的更新作为计算机技术进步和划代的主要标志。

*1.2.2 第一代计算机

第一代计算机(20 世纪 40 年代中到 20 世纪 50 年代末)为电子管计算机,其逻辑元件

采用电子管，存储器件为声延迟线或磁鼓，典型逻辑结构为定点运算。计算机“软件”一词尚未出现，编制程序所用工具为低级语言。电子管计算机体积大、速度慢（每秒千次或万次）、存储器容量小。典型机器除上述的ENIAC外，还有EDVAC、EDSAC等。第一台计算机ENIAC没有采用二进制操作和存储程序控制，不具备现代电子计算机的主要特征。

1945年3月，冯·诺依曼领导的小组发表了“存储程序（stored-program）”方式的电子数字计算机方案EDVAC，宣告了现代计算机结构思想的诞生。“存储程序”方式的基本思想是：必须将事先编好的程序和原始数据送入主存后才能执行程序，一旦程序被启动执行，计算机能在不需操作人员干预下自动完成逐条指令取出和执行的任务。

冯·诺依曼及其同事在普林斯顿高级研究院（the Institute for Advance Study at Princeton，IAS）于1946年开始设计“存储程序”计算机，该机被称为IAS计算机。

IAS计算机基本结构如图1.1所示，包含5个部件：算术逻辑单元（即运算器）、程序控制器、主存储器、输入设备和输出设备。整个计算机以“存储器”为中心，程序和数据首先输入到运算器，然后保存到主存。程序运行时，从主存调出指令，在程序控制器和运算器中执行，执行的中间结果送主存保存，最终结果送输出设备。

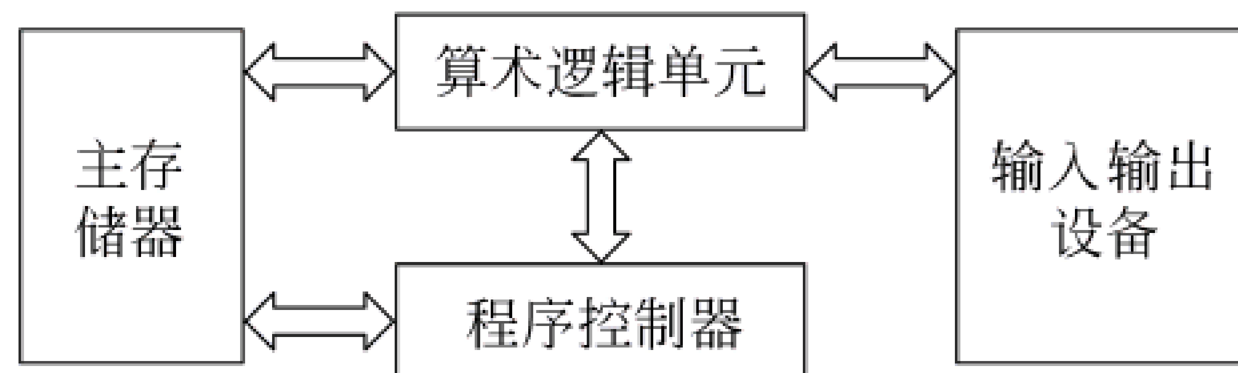


图1.1　IAS计算机基本结构

IAS计算机采用的是冯·诺依曼结构，它是后来通用计算机的原型，其基本思想主要包括以下内容。

（1）采用“存储程序”工作方式。

（2）计算机由运算器、控制器、存储器、输入设备和输出设备5个基本部件组成。

（3）各基本部件的功能如下。存储器不仅能存放数据，而且也能存放指令，形式上数据和指令没有区别，但计算机应能区分它们；控制器应能自动执行指令；运算器应能进行加、减、乘、除4种基本算术运算，并且也能进行逻辑运算；操作人员可以通过输入输出设备使用计算机。

（4）计算机内部以二进制形式表示指令和数据；每条指令由操作码和地址码两部分组成，操作码指出操作类型，地址码指出操作数的地址；由一串指令组成程序。

英国剑桥大学的M. V. Wilkes在EDVAC方案启发下，于1949年制造成功的EDSAC成为世界上第一台“存储程序”式的现代计算机，而IAS计算机直到1951年才告完成。此外，尚有1951年的UMVAC-1和1956年的IBM 704等也都属于第一代计算机。

*1.2.3　第二代计算机

第二代计算机（20世纪50年代中后期到20世纪60年代中）为晶体管计算机。1947年，美国贝尔实验室的三位科学家William Shockley、John Bardeen、Walter Brattain发明了晶体管，为计算机的发展提供了新的技术基础。该实验室于1954年研制了晶体管计算机TRADIC，而麻省理工学院于1957年完成的TX-2对晶体管计算机的发展起了重要作用。IBM公司于1955年宣布的全晶体管计算机7070和7090，开始了第二代计算机蓬勃发展的新时期，特别是1959年IBM推出的商用机IBM 1401，更以其小巧价廉和面向数据处理的特性而获得广大用户的欢迎，从而促进了计算机工业的迅速发展。

这一代计算机除了逻辑元件采用晶体管以外，其内存采用磁芯存储器，外存采用磁鼓与磁带存储器，实现了浮点运算，并在系统结构方面提出了变址、中断、I/O 处理器等新概念。这时计算机软件也得到了发展，出现了多种高级语言及其编译程序。和第一代电子管计算机相比，第二代晶体管计算机体积小、速度快、功耗低、可靠性高。

*1.2.4 第三代计算机

第三代计算机(20 世纪 60 年代中到 20 世纪 70 年代中后期)为集成电路计算机。1958 年德州仪器公司的工程师 Jack Kilby 和仙童半导体公司的工程师 Robert Noyce 几乎同时各自独立发明了集成电路，为现代计算机的发展奠定了革命性的基础，使得计算机的逻辑元件与存储器均可由集成电路实现。集成电路的应用是微电子与计算机技术相结合的一大突破，为构作运算速度快、价格低、容量大、可靠性高、体积小、功耗低的各类计算机提供了技术条件。1964 年 IBM 公司宣布了世界上第一个采用集成电路的通用计算机 IBM 360 系统研制成功，该系统的发布是计算机发展史上具有重要意义的事件。该系统采用了一系列计算机新技术，包括微程序控制、高速缓存、虚拟存储器和流水线技术等；一次就推出了 6 种机型，它们相互兼容，可广泛应用于科学计算、数据处理等领域；在软件方面首先实现了操作系统，具有资源调度、人机通信和输入输出控制等功能。IBM 360 系列的诞生，对计算机的普及应用和大规模工业化生产产生了重大影响，到 1966 年底，其产量已达到每月 400 台，5 年内总产量超过 33 000 台。

这一时期还出现了另外一个重要特点：大型/巨型机与小型机同时发展。1964 年的 CDC 6600 及随后的 CDC 7600 和 CYBER 系列是大型机代表；巨型机有 CDC STAR-100 和 64 个单元并行操作的 ILLIAC Ⅳ 阵列机等；同时小型计算机也得到很大发展，典型的有 DEC 公司的 PDP 系列。

*1.2.5 第四代计算机

第四代计算机(20 世纪 70 年代后期开始)为超大规模集成电路计算机。20 世纪 70 年代初，微电子学飞速发展而产生的大规模集成电路和微处理器给计算机工业注入了新鲜血液。其后，大规模(LSI)和超大规模(VLSI)集成电路成为计算机的主要器件，其集成度从 20 世纪 70 年代初的几千个晶体管/片(如 Intel 4004 为 2000 个晶体管)到 20 世纪末的千万个晶体管/片。以最大的微处理器制造商 Intel 产品为例(见表 1.1)可以看出，半导体集成电路的集成度越来越高，速度也越来越快，其发展遵循摩尔定律：由于硅技术的不断改进，每 18 个月，集成度将翻一番，速度将提高一倍，而其价格将降低一半。戈登·摩尔(Golden Moore)是 Intel 公司的创始人之一，摩尔定律是 1965 年美国《电子》杂志的总编在采访摩尔先生时他对半导体芯片工业发展前景的预测，40 多年来的实践证明，摩尔定律的预测是基本准确的。

摩尔定律对计算机工业的发展具有以下重要意义。

(1) 由于定律预测了半导体产品和技术每经过一年半时间将会翻一番，因此如果发展速度慢于这个定律的指标，那么将有被淘汰的危险，“逆水行舟，不进则退”，这就迫使计算机芯片制造商不断改进技术，提高性能。

表 1.1 Intel CPU 芯片性能比较

芯片名	年代	集成度（晶体管数）	主频范围（MHz）	线宽（μm）	数据总线（位）	地址总线（位）
4004	1971	2300	0.74	2	4	
8080	1974	8000	4	1.5	8	16
8086	1978	2.9 万	4.77	1.5	16	20
80286	1984	13.4 万	6～25	1～1.5	16	24
80386	1985	27.5 万	20～33	1～1.5	32	32
80486	1989	118.5 万	33～100	1	32	32
Pentium	1993	300 万	60～133	0.6	64	32
Pentium Pro	1995	550 万	150～233	0.6	64	32
Pentium Ⅱ	1997	750 万	233～400	0.35	64	32
Pentium Ⅲ	1999	950 万	450～1000	0.25	64	32
Pentium 4	2000	4200 万	1500～3800	0.18	64	32

(2) 芯片价格的持续下降，一方面迫使计算机芯片制造商采取正确的价格策略，以提高产品的竞争力；另一方面也为计算机的普及创造了有利条件。

(3) 定律不仅适用于“硬件”，同时也驱动着软件工业和市场的发展。由于硬件性能的不断改进和提高，软件也必须适应硬件的发展而进行不断的修改与创新，否则，也将面临被淘汰的危险。微软公司总裁比尔·盖茨曾经不断地以下面一句话来鞭策下属：“微软离破产永远只有一年半时间！”

随着超大规模集成电路与微处理器技术的长足进步和现代科学技术对提高计算能力的强烈要求，并行处理技术的研究与应用以及众多巨型机的产生也成为这一时期计算机发展的特点。1976 年，Cray 公司推出的 Cray-1 向量巨型机，具有 12 个功能部件，运算速度达每秒 1.6 亿次浮点运算。不少巨型机采用成百上千个高性能处理器组成大规模并行处理系统，其峰值速度已达到每秒几千亿或万亿次，这种并行处理技术成为 20 世纪 90 年代巨型机发展的主流。

第四代计算机时期的另一个重要特点是计算机网络的发展与广泛应用。由于计算机技术与通信技术的高速发展与密切结合，掀起了网络热潮，大量的计算机联入不同规模的网络中，然后通过 Internet 与世界各地的计算机相联，大大扩展和加速了信息的流通，增强了社会的协调与合作能力，使计算机的应用方式也由个人计算方式向网络化方向发展。

由于计算机在技术发展和使用方式上的不断进步，目前学术界和工业界大多已不再沿用传统以元器件划分“第 x 代计算机”的提法。

60 多年来，人们使用计算机的方式发生了巨大变化。早先是多人共享一台计算机(分时计算方式)，然后是一人使用一台计算机(个人计算方式)，进而发展到目前多人使用多台计算机的网络计算方式。进一步的发展趋势是人们将进入普适计算(pervasive computing)时代。

计算机技术的另一个重要发展方向是计算机的智能化，其着眼点是发展以知识为基础

的智能化处理能力,用以模拟或部分替代人的智能活动,并提供智能化的人机交互接口,使计算机具有自然的人机通信能力。普适化、智能化、嵌入式和网络化将是未来计算机发展的主要方向。

1.3 计算机系统的组成

计算机系统由硬件和软件两部分组成。硬件是具体物理装置的总称,人们看到的各种芯片、板卡、外设、电缆等都是计算机硬件。软件包括运行在硬件上的程序和数据以及相关的文档。程序是指挥计算机如何操作的一个指令序列,数据是指令操作的对象。

1.3.1 计算机硬件

在计算机问世初期,"计算机"一词实际上只指计算机硬件。直到 20 世纪 60 年代,由于程序设计技术的进步,才引进了计算机硬件和软件的概念。

计算机硬件主要包括中央处理器、存储器、I/O 控制器、外部设备和各类总线等。中央处理器(Central Processing Unit)简称为 CPU 或处理器,是整个计算机的核心部件,主要用于指令的执行。CPU 中最基本的部分主要包含两个部件:数据通路和控制器。数据通路主要用来执行算术和逻辑运算以及寄存器和存储器的读写操作等。控制器用来对指令进行译码,生成相应的控制信号,以控制数据通路进行正确的操作。存储器分为内存和外存,内存包括主存(Main Memory,MM)和高速缓存(cache)。因为早期计算机中没有高速缓存,因此传统上内存和主存没有差别,所以一般情况下也并不区分内存和主存。输入输出(Input/Output)简称为 I/O,I/O 设备通过 I/O 控制器或 I/O 适配器连到主机上,I/O 控制器或 I/O 适配器统称为 I/O 接口或 I/O 模块。CPU、主存和 I/O 模块通过总线互连。

图 1.2 是一个典型的多总线计算机系统的硬件结构示意图。

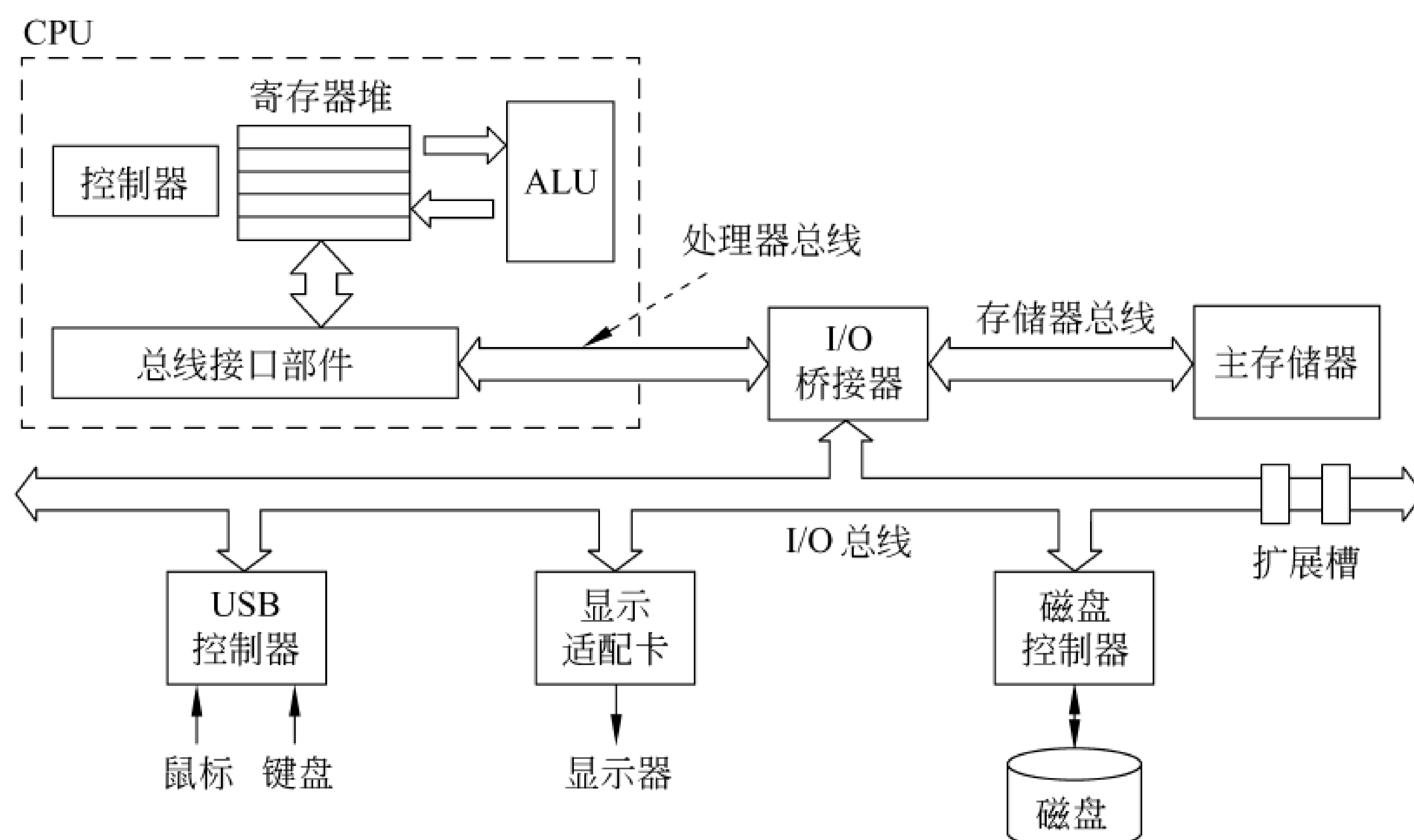


图 1.2 一个典型计算机系统的硬件组成

如图 1.2 所示,CPU 中包含控制器(controller)、算术逻辑部件(Arithmetic Logic Unit,ALU)、寄存器堆(register file,也称通用寄存器组或寄存器文件)、总线接口部件等,CPU 通

过处理器总线、I/O 桥接器与主存储器和输入输出设备交换信息；主存通过存储器总线、I/O 桥接器与 CPU 和输入输出设备交换信息，存储控制器可以做在 I/O 桥接器中；I/O 设备通过各自的外设控制器或适配器连到 I/O 总线上，例如，可以把鼠标和键盘连接到 USB 控制器的接口上，显示器连接到显示适配卡的接口上等。

ALU 是数据处理部件，用于执行数据的算术和逻辑运算，ALU 处理的数据来自寄存器堆；磁盘和主存是数据存储部件，分别用于存储长期保存信息和临时保存信息；各类总线以及总线接口部件、I/O 桥接器、I/O 扩展槽、I/O 控制器和显示适配器等都是互连部件，用于完成数据传送任务。所有这些部件的操作都通过 CPU 中的控制器执行指令来完成。

从外部来看，普通台式个人计算机(PC)是用各种电缆将显示器、键盘、鼠标和机箱等连接而成的一个装置。打开一台普通台式机的机箱后，看到的是如图 1.3 所示的一组电路板、芯片和连线，如主板、电源、风扇和硬盘驱动器等。

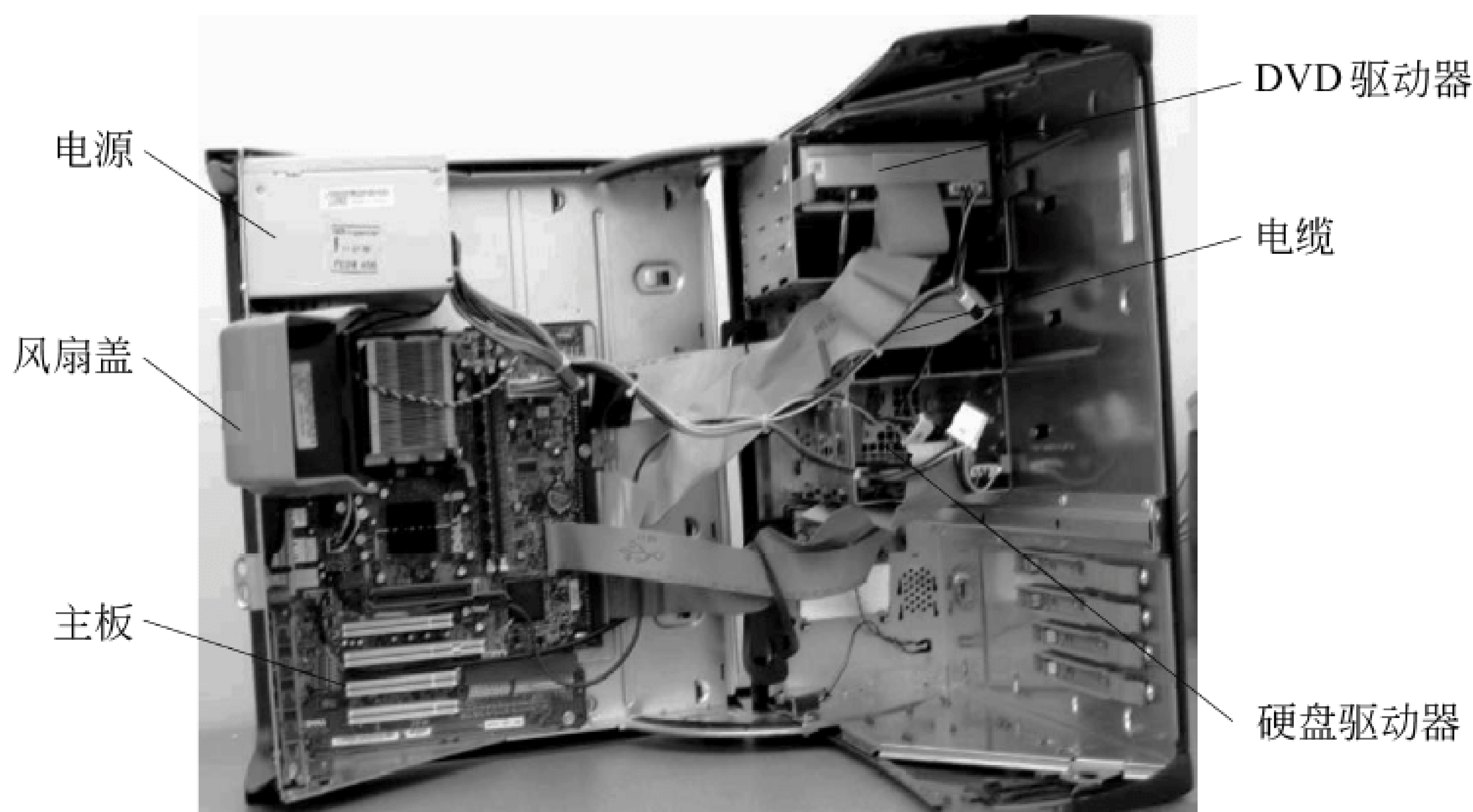


图 1.3　台式个人计算机机箱内的部件

图 1.4 所示是个人计算机主板，其中有一个处理器芯片插座，用于插入相应的 CPU 芯片，通过 PCI 总线插槽可连接相应外设，内存条也可插入插槽进行扩充和更换。

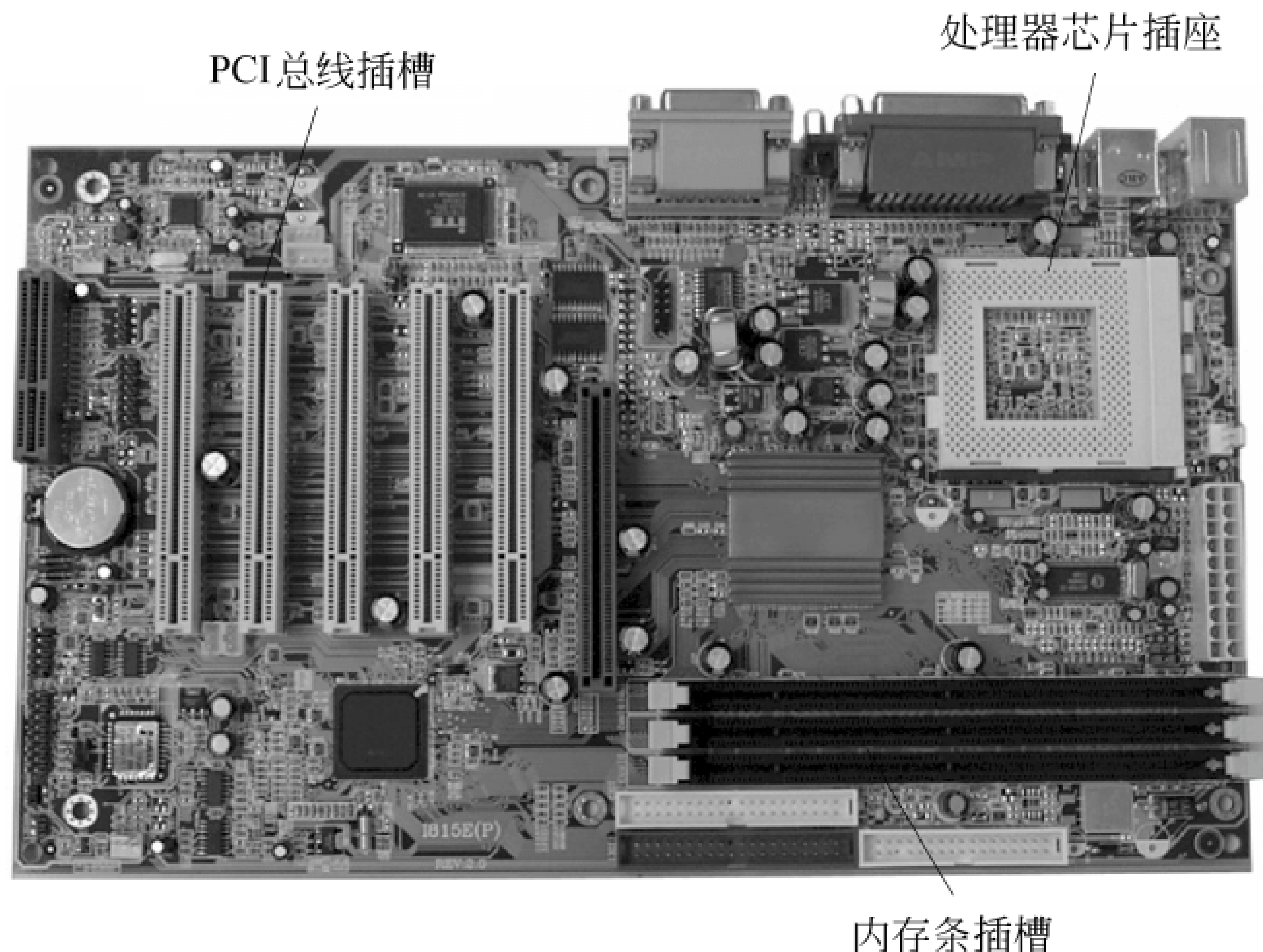


图 1.4　台式个人计算机主板

图 1.5 是对计算机硬件进行解剖的示意图，显示了一台个人计算机的硬件结构分解过程。计算机主机由多个电路板用总线连接而成，每个电路板上又焊接了多个集成电路芯片，每个芯片中有十几个电路模块，每个模块中有上千万个单元，每个单元中有几个门电路，每个门电路实现基本的逻辑运算，因为计算机中所有信息都采用二进制编码表示，二进制的 1 和 0 对应逻辑值“真”和“假”，因此，可以方便地通过逻辑运算电路来实现算术运算。

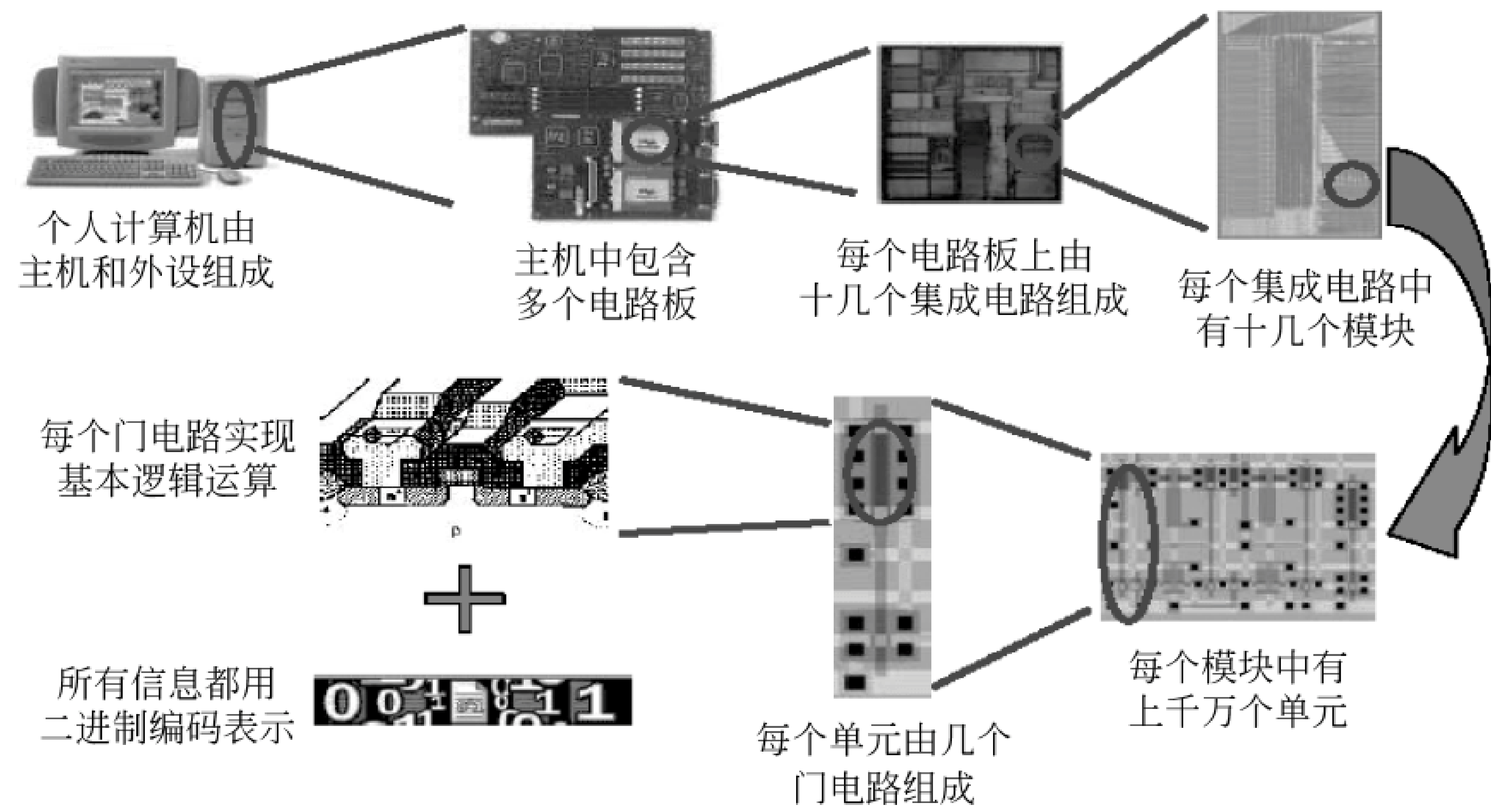


图 1.5　个人计算机的硬件结构解剖

图 1.6 是 Pentium 4 处理器芯片的内部结构示意图。左边是芯片的显微照片，右边是功能模块。Pentium 4 处理器芯片中除了整数运算数据通路外，还集成了浮点运算数据通路和多媒体处理数据通路；Pentium 4 处理器将 L1 cache（包括指令 cache 和数据 cache）和 L2 cache 都做在了 CPU 芯片内。此外，还有具有支持高级流水线和超线程的部件以及用于 I/O 访问和存储器访问的接口部件。

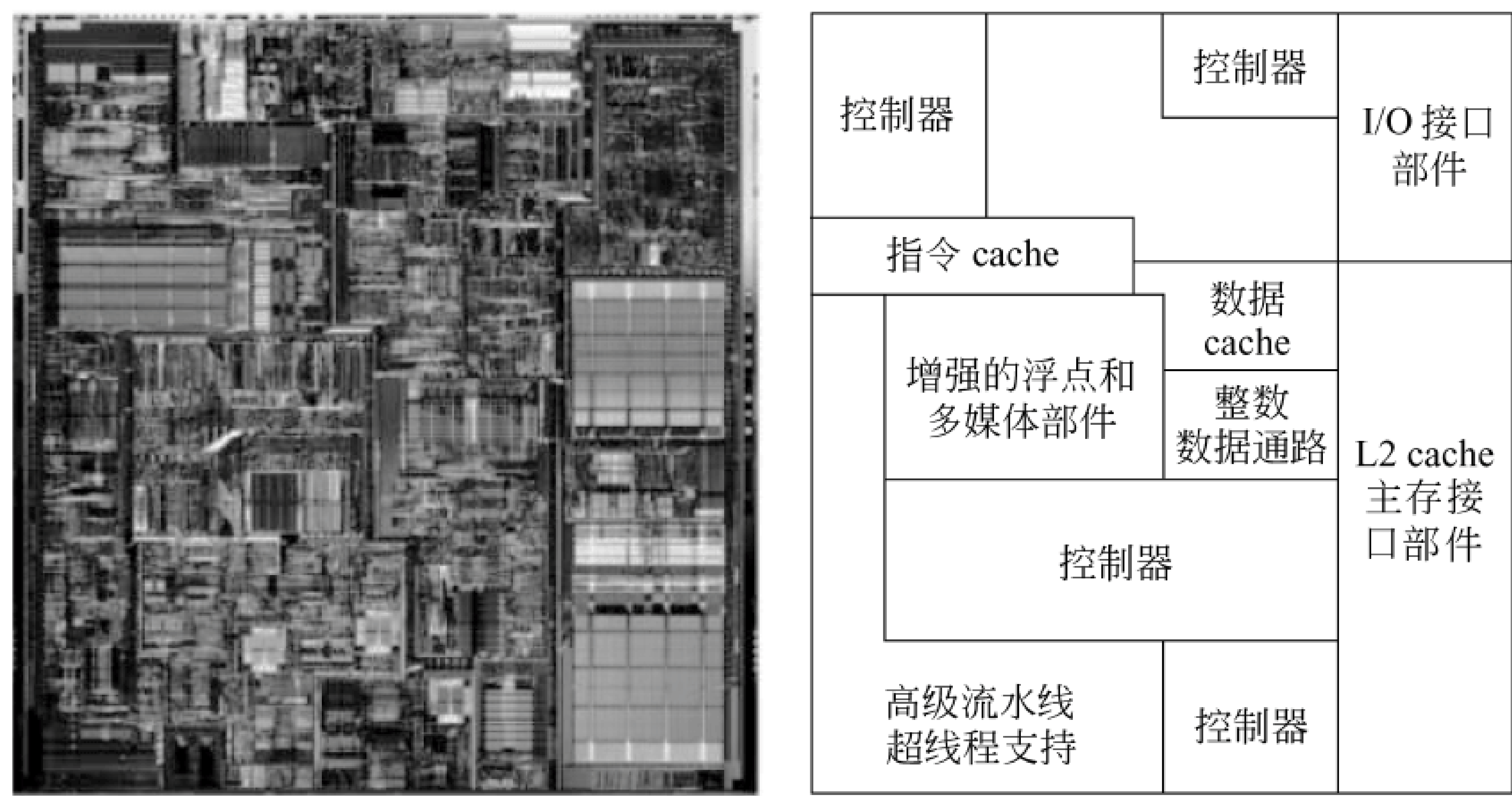


图 1.6　Pentium 4 处理器芯片的内部组成

从20世纪40年代计算机诞生以来，尽管硬件技术已经经历了4个发展阶段，计算机体系结构也已经取得了很大的发展，但绝大部分计算机的硬件基本组成仍然具有冯·诺依曼结构计算机的特征。

1.3.2 计算机软件

计算机的工作由存储在其内部的程序控制，这是冯·诺依曼结构计算机“存储程序”工作方式的重要特征，因此程序或者软件质量的好坏将大大影响计算机性能的发挥。

软件的发展受计算机硬件和计算机应用的推动和制约，其发展过程大致分三个阶段。

从第一台计算机上的第一个机器代码程序出现到实用的高级语言出现为第一阶段(1946—1956年)。这时期的计算机应用以科学计算为主，计算量较大，但输入输出量不大；机器以CPU为中心，存储器较小；直接采用机器语言编程，因而程序设计与编制工作复杂、繁琐、易出错。这时尚未出现软件一词。

从实用的高级程序设计语言出现到软件工程概念出现以前这段时间为第二阶段(1956—1968年)。这时期除了科学计算外，还出现了大量数据处理问题，计算量不大，但输入输出量较大。机器结构转向以存储器为中心，出现了大容量存储器，输入输出设备增加，软件概念也开始出现。为了充分利用处理器、存储器和输入输出等计算机资源，出现了操作系统；为了提高编程工作效率，出现了高级语言；为了适应大量的数据处理，出现了数据库及其管理系统。随着软件规模和复杂性的不断提高，软件开发过程中，甚至出现了人们难以控制的局面，即所谓软件危机。为了克服这种危机，人们研究和采用了很多技术方法，这就导致了“软件工程”概念和方法的出现。

软件工程出现以后至今一直处于第三阶段。对于一些复杂的大型软件，基于个人和简单团队分工的传统开发方式进行开发不仅效率低、可靠性差，且很难完成，必须采用工程方法才能实现。为此，从20世纪60年代末开始，软件工程技术得到了迅速的发展，出现了“计算机辅助软件设计”、“软件自动化”等技术方法和实验系统。目前，人们除了研究改进软件开发技术外，还着重研究具有智能化、自动化、集成化、并行化以及自然化特征的软件新技术。

根据软件的用途，一般将软件分成系统软件和应用软件两大类。系统软件包括为有效、安全地使用和管理计算机以及为开发和运行应用软件而提供的各种软件，介于计算机硬件与应用程序之间，它与具体应用关系不大。系统软件包括操作系统(如Windows)、语言处理系统(如C语言编译器)、数据库管理系统(如Oracle)和各类实用程序(如磁盘碎片整理程序、备份程序)。操作系统主要用来管理整个计算机系统的硬件资源，包括对它们进行调度、管理、监视和服务等，操作系统还提供计算机用户和硬件之间的人机交互界面，并提供对应用软件的支持；语言处理系统提供一个用高级语言编程的环境，包括源程序编辑、翻译、调试、连接、装入运行等功能；数据库管理系统是一种用来建立、使用和维护数据库的软件系统。

专门为数据处理、科学计算、事务管理、多媒体处理、工程设计以及过程控制等应用所编写的各类程序都称为应用软件，例如，人们平时经常使用的电子邮件收发软件、播放软件、游戏软件、炒股软件、文字处理软件、电子表格软件、演示文稿制作软件等都是应用软件。

1.4 计算机系统的层次化结构

按照在计算机上完成任务的不同，可以把使用计算机的用户分成以下 4 类：最终用户、系统管理员、应用程序员和系统程序员。

使用应用程序完成特定任务的计算机用户称为最终用户(end user)。大多数计算机使用者都属于最终用户。例如，使用炒股软件的股民、玩计算机游戏的人、进行会计电算化处理的财会人员等。

系统管理员(system administrator)是指利用操作系统等软件提供的功能对系统进行配置、管理和维护以建立高效合理的系统环境供计算机用户使用的操作人员。其职责主要包括安装、配置和维护系统的硬件和软件，建立和管理用户账户，升级软件，备份和恢复业务系统和数据等。

应用程序员(application programmer)是指使用高级编程语言编制应用软件的程序员；而系统程序员(system programmer)则是指设计和开发系统软件的程序员，如开发操作系统、编译器、数据库管理系统等系统软件的程序员。

计算机系统采用层次化体系结构，不同用户工作在不同层次，他们所看到的计算机系统是不一样的。

1.4.1 最终用户眼中的计算机

早期的计算机非常昂贵，只能由少数专业化人员使用。随着 20 世纪 80 年代初个人计算机的迅速普及以及 20 世纪 90 年代初多媒体计算机的广泛应用，特别是互联网技术的发展，计算机已经成为人们日常生活中的重要工具。人们利用计算机播放电影、玩游戏、炒股票、发邮件、查信息、聊天打电话等，计算机的应用无处不在。因而，许多普通人都成了计算机的最终用户。

计算机最终用户使用键盘和鼠标等外设与计算机交互，通过操作系统提供的用户界面启动执行应用程序或系统命令，从而完成用户任务。因此，最终用户能够感知到的只是系统提供的简单人机交互界面和安装在计算机中的相关应用程序。

1.4.2 系统管理员眼中的计算机

相对于普通的计算机最终用户，系统管理员作为管理和维护计算机系统的专业人员，对计算机系统的了解要深入得多。系统管理员必须能够安装、配置和维护系统的硬件和软件，能建立和管理用户账户，需要时能升级硬件和软件，备份和恢复业务系统和数据等。也就是说，系统管理员应该非常熟悉操作系统提供的有关系统配置和管理方面的功能，很多普通用户解决不了的问题，系统管理员必须能够解决。

因此，系统管理员能感知到的是系统中部分硬件层面，软硬件配置和系统管理层面以及相关的实用程序和人机交互界面。

1.4.3 应用程序员眼中的计算机

应用程序员大多使用高级程序设计语言编写程序。所谓高级程序设计语言(high level programming language)是指面向算法设计的较接近于日常所用的英语书面语言的程序设计语言,例如 BASIC、C、Fortran、Java 等。

应用程序员所看到的计算机系统除了计算机硬件、操作系统提供的编程接口(API)、人机交互界面和实用程序外,还包括相应的程序语言处理系统。

在语言处理系统中,除了翻译程序外,通常还包括编辑程序、链接程序、装入程序以及将这些程序和工具集成在一起所构成的集成开发环境(Integrated Development Environment,IDE)等。此外,语言处理系统中还包括可供应用程序调用的各类函数库。

1.4.4 系统程序员眼中的计算机

系统程序员开发操作系统、编译器和实用程序等系统软件时,需要熟悉计算机底层的相关硬件和系统结构,甚至可能需要直接与计算机硬件和指令系统打交道。例如,直接对各种控制寄存器、用户可见寄存器、I/O 接口等硬件进行控制和编程。因此,系统程序员必须熟悉指令系统、机器结构和相关的机器功能特性,有时还要直接用汇编语言等低级程序设计语言编写程序代码。

通常把系统程序员感觉到的计算机的功能特性和概念性结构称为计算机系统结构或指令集体系结构(Instruction Set Architecture,ISA)。其内容包括:数据类型及格式,指令系统及指令格式,寻址方式和可访问空间大小,程序可访问的寄存器个数、位数和编号,控制寄存器的定义,I/O 空间的编址方式,中断结构,机器工作状态的定义和切换,输入输出连接结构和数据传送方式,存储保护方式等。这些都是系统程序员为了使编写的系统软件能在机器上正确执行而需要了解和遵循的计算机属性。

1.4.5 程序开发与执行过程

程序的开发和执行涉及计算机系统的各个不同层面,因而计算机系统层次化结构的思想体现在程序开发和执行过程的各个环节中。下面以简单的 hello 程序为例,介绍该程序的开发与执行过程,以便加深对计算机系统层次化结构概念的认识。

以下是 hello.c 的 C 语言源程序代码。

```
1 #include <stdio.h>
2
3 int main()
4 {
5 printf("hello, world\n");
6 }
```

为了让计算机能执行上述应用程序,应用程序员应按照以下步骤进行。

(1) 通过程序编辑软件得到 hello.c 文件。hello.c 在计算机中以 ASCII 字符方式存放,如图 1.7 所示,图中给出了每个字符对应的 ASCII 码的十进制值。

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

图 1.7　hello.c 源程序文件的表示

（2）将 hello.c 进行预处理、编译、汇编和链接，最终生成可执行代码文件，其过程如图 1.8 所示。

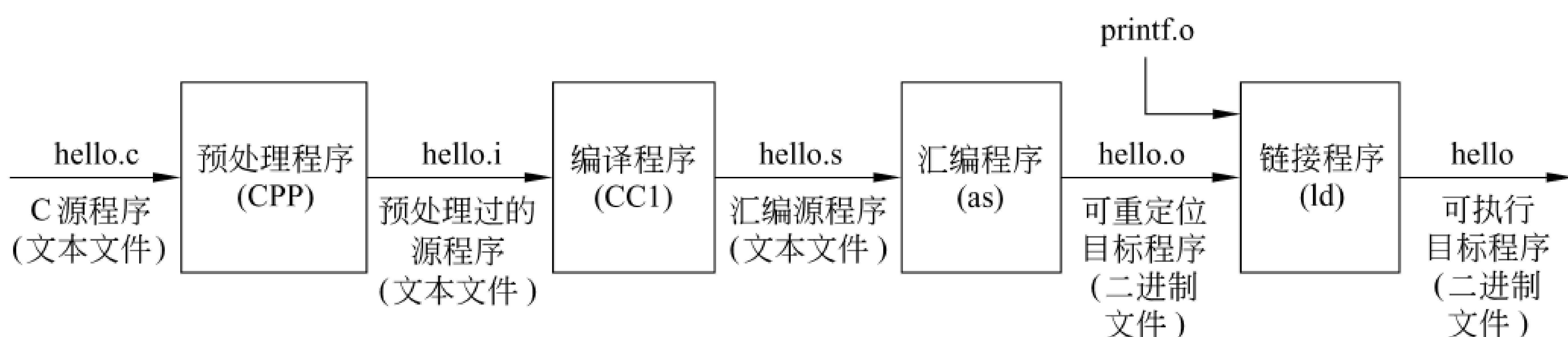


图 1.8　hello.c 源程序文件到可执行代码文件的转换过程

计算机的控制器自动执行的是指令，每条指令由操作码和地址码两部分组成，操作码指出操作类型，地址码指出操作数的地址。执行程序实际上是执行一个指令序列。也就是说，不管用什么高级语言编写的程序都必须转换为一个指令序列才能在计算机上执行。

通常把计算机能直接执行的指令的集合称为指令集(instruction set)，也称为机器语言(machine language)。指令由二进制编码构成，因此由指令构成的程序文件是二进制文件(binary file)，例如，图 1.8 中的可重定位目标文件 hello.o、printf.o 和可执行文件 hello 都是二进制文件。

可重定位目标文件由单独的源程序文件转换得到，每个目标文件中指令和数据所用的地址都在各自独立的地址空间中，链接程序将关联的多个目标程序连接生成可执行文件时，通过重定位指令和数据来将多个目标文件结合为一个整体，共享一个共同的地址空间。

机器语言用二进制进行编码，每条机器指令都是一个 0/1 序列，因此，可读性很差，也不易记忆，给程序员读写程序带来极大的困难。因此，人们引入了一种机器语言的符号表示语言，通过用简短的英文符号和二进制代码建立对应关系，以方便程序员编写和阅读机器语言程序，这种语言被称为汇编语言(assembly language)。机器语言和汇编语言都是面向计算机硬件的、与计算机系统结构密切相关的低级语言。

任何高级语言源程序和汇编语言源程序都必须转换为机器语言程序才能被计算机执行，我们把进行这种转换的软件统称为“程序设计语言处理系统”。通常，应用程序员和系统程序员都借助“程序设计语言处理系统”来开发软件。

任何一个语言处理系统中，都包含一个翻译程序(translator)，它能把一种编程语言表示的程序转换为等价的另一种编程语言程序。被翻译的语言和程序分别被称为源语言和源

程序，翻译生成的语言和程序分别被称为目标语言和目标程序。通常把用 ASCII 码字符或汉字字符表示的文件称为文本文件(text file)，一般源程序文件都是文本文件，如图 1.8 中的 hello.c、hello.i 和 hello.s 文件。

翻译程序有以下三类。

(1) 汇编程序(assembler)：也称汇编器，用来将汇编语言源程序翻译成机器语言目标程序。

(2) 解释程序(interpreter)：也称解释器，用来将源程序中的语句按其执行顺序逐条翻译成机器指令并立即执行。例如，BASIC 解释程序直接启动 BASIC 源程序执行，不生成目标程序。

(3) 编译程序(compiler)：也称编译器，用来将高级语言源程序翻译成汇编语言或机器语言目标程序。

图 1.9 给出了实现两个相邻数组元素交换功能的不同层次语言的描述。从中可以看出，在高级语言程序中，可直观地用 3 个赋值语句实现；在经编译后生成的汇编语言程序中，可用 4 个汇编语句表示，有两条装入(lw)指令和两条存储(sw)指令；在经汇编后生成的机器语言程序中，对应的机器指令是特定格式的二进制代码，计算机能直接识别和执行，执行时通过控制器将指令操作码进行译码解释成控制信号(control signal)来控制数据通路执行。

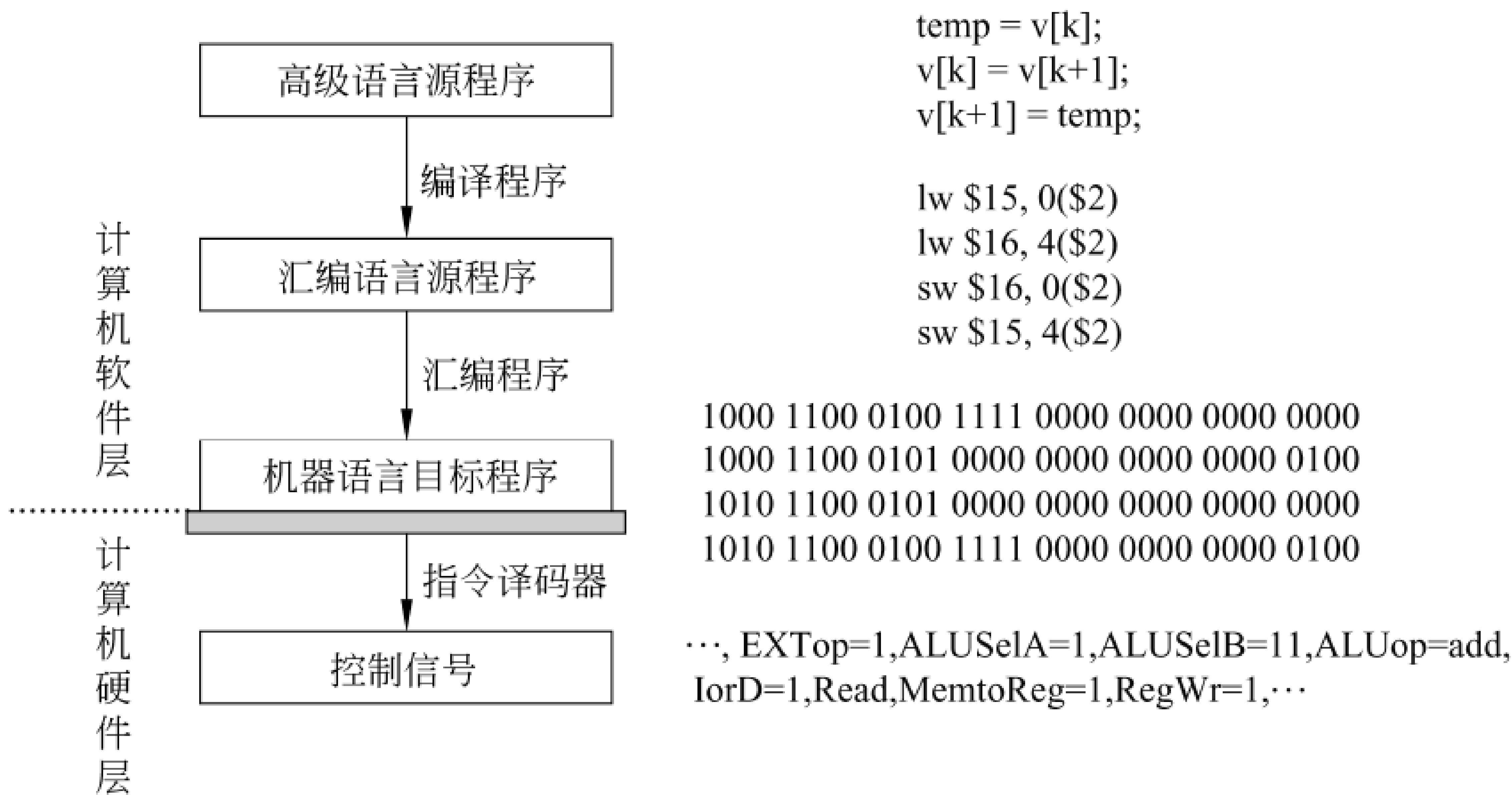


图 1.9　不同层次语言之间的等价转换

在计算机技术中，一个存在的事物或概念从某个角度看似不存在，称为透明性现象。通常，在一个计算机系统中，系统程序员所看到的底层机器级的概念性结构和功能特性对高级语言程序员(通常就是应用程序员)来说是透明的，也即看不见的。因为对应用程序员来说，他们直接用高级语言编程，不需要了解有关汇编语言的编程问题，也不用了解机器语言中规定的指令格式、寻址方式、数据类型和格式等指令系统方面的问题。

一个计算机系统可以认为是由各种硬件和各类软件采用层次化方式构建的分层系统，不同用户工作在不同的系统结构层，所看到的计算机的概念性结构和功能特性是不同的。综上所述，不同计算机用户工作所在的系统结构层如图 1.10 所示。

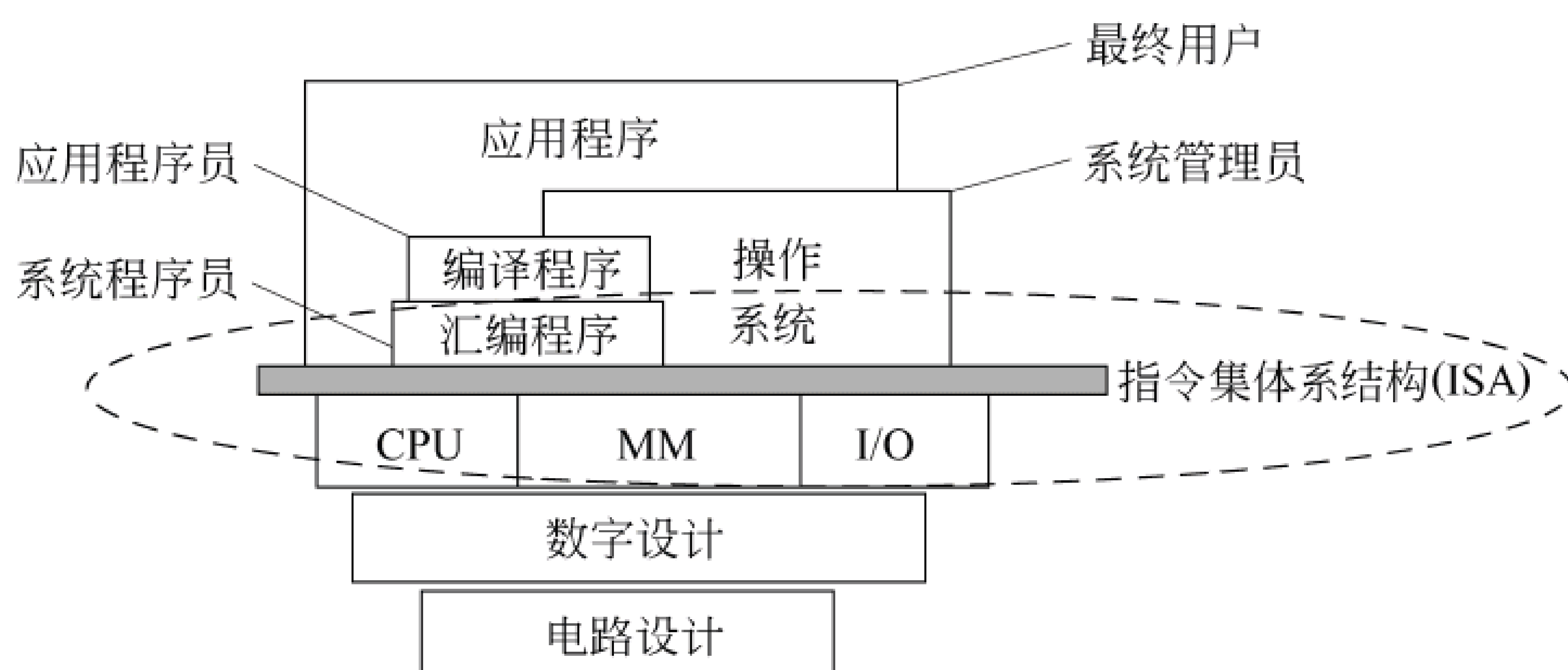


图 1.10　计算机系统的层次化结构

从图 1.10 中可看出，ISA 处于硬件和软件的交界面上，硬件所有的功能都由 ISA 集中体现，软件通过 ISA 在计算机上执行。所以，ISA 是整个计算机系统中的核心部分。

ISA 层下面是硬件部分，上面是软件部分。硬件部分包括 CPU、主存和输入输出等主要功能部件，这些功能部件通过数字逻辑电路设计实现。软件部分包括低层的系统软件和高层的应用软件，汇编程序、编译程序和操作系统等这些系统软件直接在 ISA 上实现，系统程序员所看到的机器的属性是属于 ISA 层面的内容，所看到的机器是配置了指令系统的机器，称为机器语言机器，工作在该层次的程序员称为机器语言程序员；系统管理员工作在操作系统层，所看到的是配置了操作系统的虚拟机器，称为操作系统虚拟机；汇编语言程序员工作在提供汇编程序的虚拟机器级，所看到的机器称为汇编语言虚拟机；应用程序员大多工作在提供编译器或解释器等翻译程序的语言处理系统层，因此，应用程序员大多用高级语言编写程序，因而也称为高级语言程序员，所看到虚拟机器称为高级语言虚拟机；最终用户则工作在最上面的应用程序层。

1.5　本教材的主要内容和组织结构

图 1.10 中虚线所框部分是本教材的主要内容，其中最核心的部分是指令集体系结构(ISA)，其他所有内容都围绕 ISA 展开。具体来说，本教材的主要内容和组织结构说明如下。

1. 计算机系统的性能评价

本教材在第 1 章的第 1.6 节首先介绍计算机系统性能评价方法，以后每章的内容都围绕 ISA 设计时在计算机组织和结构方面要考虑的问题展开。计算机设计的目标是提高计算机系统的性能，掌握了性能评价方法，就可对不同的解决问题方法的优劣用性能指标来进行定量的比较和评估。

2. 数据的机器级表示

指令处理的对象是数据。从不同的角度来看，计算机处理的数据和处理方式有不同的表现形式。比如，从外部数据形式来看，计算机可处理数值、文字、图、声音、视频，甚至各种模拟信息量；从算法描述的角度来看，有图、表、树、队列、矩阵等结构类型的数据；从高级语言程序员的角度来看，有数组、结构、指针、实数、整数、布尔数、字符和字符串等类型的数据。但是，不管以什么形式出现，在计算机内部这些数据最终都要由机器指令来处理。从 ISA 设计的角度来看，只需关心数据在计算机中底层的机器级表示。

第 2 章重点讨论计算机内部数据的机器级表示方式，包括无符号整数、有符号整数、浮

点数、西文字符和汉字、十进制数的表示，各种类型数据的转换，数据的宽度和存放顺序以及几种常用检/纠错码的编码表示与使用方法。

3. 运算方法和运算部件

计算机的功能通过执行程序来实现，任何程序最终都转换为机器指令。指令中包含的各种算术逻辑运算能直接在硬件上执行，执行这些运算的硬件称为运算部件。

第3章重点讨论整数和浮点数的加、减、乘、除等运算方法以及运算部件。首先分析高级语言和机器指令中涉及的各类运算，然后介绍这些运算用到的核心部件——算术逻辑单元（ALU）的组成与工作原理，在此基础上，对这些运算在计算机内部的实现算法和过程进行详细说明，最后介绍实现这些运算的运算部件。

4. 存储器层次结构

计算机采用"存储程序"的工作方式，程序和数据存放在存储器中。有了存储器，从而能使计算机自动地从存储器中取出一条条指令执行，执行时从存储器中取出相应的数据，执行后将结果送存储器保存。

为了使存储器在容量、速度和价格上达到一定的要求，计算机内部用多种不同类型的存储器组成了一种层次结构的存储器系统。

第4章主要介绍构成层次化存储器系统的几类存储器的工作原理和组织形式。主要包括半导体随机访问存储器的组织，只读存储器和Flash存储器，高速缓冲存储器以及虚拟存储系统等。

5. 指令系统

一台计算机能执行的机器指令全体称为该机的指令集或指令系统，它是构成程序的基本元素，也是硬件设计的依据，它衡量机器硬件的功能，反映硬件对软件支持的程度。指令系统是ISA中最核心的部分，ISA设计的好坏直接决定计算机的性能和成本，因此，指令系统的设计至关重要。

第5章主要介绍指令系统设计中涉及的各个技术方面，主要包括指令格式、操作类型、操作数类型、寻址方式、操作码编码、指令系统的风格以及程序的机器级表示等。

6. 中央处理器

计算机的所有功能都通过执行程序来实现，而程序的执行过程就是自动地执行一条条指令的过程，用来实现该过程的主要部件是CPU，因此，CPU是计算机的核心硬件。

根据执行一条指令所花时钟数的多少来分，可将CPU分为单周期处理器、多周期处理器和流水线处理器；根据CPU中控制器的实现方式来分，可将CPU分为硬连线路控制（hardwired control）处理器和微程序控制（microprogrammed control）处理器，也有一些CPU采用硬连线路和微程序混合控制的方式实现。

第6章主要介绍CPU的基本功能和基本组成以及非流水线的单周期和多周期处理器的工作原理和设计方法。

7. 指令流水线

现代计算机基本上都采用流水线方式执行指令。指令流水线方式将一条指令的执行过程分解成若干个功能阶段，每个阶段作为一个流水段。流水线方式使得同一时刻有多条指令同时执行，因而加快了程序执行的速度。

第7章主要介绍流水线处理器的工作原理和设计方法。

8. 系统总线

现代计算机系统中,CPU、主存和 I/O 等功能部件之间采用总线方式进行互连。总线互连结构有两个主要优点：灵活和成本低。它的灵活性体现在新部件可以很容易添加到总线上,并且部件可以在使用相同总线的计算机系统之间互换。因为一组单独的连线可被多个部件共享,所以总线具有较高的性价比。

第 8 章着重介绍总线基本概念、总线裁决、总线定时、总线标准以及现代计算机的总线互连结构。

9. 输入输出组织

输入输出组织是用来控制外设与主机之间进行信息交换的机构。通常把外部设备及其接口线路、I/O 控制部件以及 I/O 设备驱动软件统称为输入输出系统,输入输出系统要解决的问题是对各种形式的信息进行输入和输出的控制。

第 9 章将重点介绍常用的外部设备,I/O 接口的功能和结构,外设的编址和寻址以及在主机和外设之间进行数据传送的各种输入输出控制方式等内容。

1.6 计算机系统性能评价

一个完整的计算机系统由硬件和软件构成,硬件性能的好坏对整个计算机系统的性能起着至关重要的作用。硬件的性能检测和评价比较困难,因为硬件的性能只能通过运行软件才能反映出来,在相同硬件上运行不同类型的软件,或者用同样的软件不同的数据集测试,所测到的性能都可能不同,因此,必须有一套综合的测试和评价硬件性能的方法。

1.6.1 计算机性能的定义

吞吐率(throughput)和响应时间(response time)是考量一个计算机系统性能的两个基本指标。吞吐率表示在单位时间内所完成的工作量。在有些场合下,吞吐率也称为带宽(bandwidth),响应时间也称为执行时间(execution time)或等待时间(latency),是指从作业提交开始到作业完成所花的时间。一个程序的响应时间除了程序包含的指令在 CPU 上执行所花的时间外,还包括磁盘访问时间、存储器访问时间、输入输出操作所需时间以及操作系统运行这个程序所花的额外开销等。

不同的应用场合计算机用户关心的性能是不同的。例如,在多媒体应用场合,用户希望音/视频的播放要流畅,即单位时间内传输的数据量要大,因而关心的是系统吞吐率是否高;而在银行、证券等事务处理应用场合,用户希望业务处理速度快,不需长时间等待,因而更关心响应时间是否短;还有些应用场合(如 ATM、文件服务、Web 服务等),用户则同时关心吞吐率和响应时间。

1.6.2 计算机性能的测试

如果不考虑应用背景而直接比较计算机性能,则大都用执行时间来衡量。因此,从执行时间来考虑,完成同样工作量所需时间最短的那台计算机性能是最好的。

操作系统在对处理器进行调度时,一段时间内往往会让多个程序(更准确地说是进程)轮流使用处理器,因此在某个用户程序执行过程中,可能同时还会有其他用户程序和操作系

统程序在执行，所以，用户感觉到的某个程序的执行时间并不是其真正的执行时间。通常把用户感觉到的执行时间分成两部分：CPU 时间和其他时间。CPU 时间指 CPU 真正用在程序执行上的时间，它又包括以下两部分：(1)用户 CPU 时间，指用来运行用户程序代码的时间；(2)系统 CPU 时间，指为了执行用户程序而需要运行操作系统程序的时间。其他时间指等待 I/O 操作完成的时间或 CPU 用在其他用户程序执行上的时间。

计算机系统的性能评价主要考虑 CPU 性能。系统性能和 CPU 性能不等价，两者有一些区别。系统性能是指系统的响应时间，它与 CPU 外的其他部分也有关系；而 CPU 性能是指用户 CPU 时间，它只包含在 CPU 上运行用户程序代码的时间。

在对 CPU 时间进行计算时需要用到以下几个重要的概念和指标。

1. 时钟周期(clock cycle，tick，clock tick，clock)

计算机执行指令的过程被分成若干步骤和相应的动作来完成，每一步动作都要有相应的控制信号进行控制，这些控制信号何时发出、作用时间多长，都要有相应的定时信号进行同步。因此，CPU 必须能够产生同步的时钟定时信号，也就是 CPU 的主脉冲信号，其宽度称为时钟周期。

2. 时钟频率(clock rate，主频)

CPU 的主频就是 CPU 中的主脉冲信号的时钟频率，是 CPU 时钟周期的倒数。

3. CPI(cycles per instruction)

CPI 表示执行指令所需的时钟周期数。由于不同指令的功能不同，所需的时钟周期数也不同，因此，对于一条特定指令而言，其 CPI 指执行该条指令所花的时钟周期数，此时 CPI 是一个确定的值；对于一个程序或一台机器来说，其 CPI 指该程序或该机器指令集中的所有指令执行所用的平均时钟周期数，此时，CPI 是一个平均值。

已知上述参数或指标，可以通过以下公式来计算用户程序的 CPU 时间。

$$\begin{aligned}\text{CPU 执行时间} &= \text{程序所含时钟周期数} \div \text{时钟频率}\\ &= \text{程序所含时钟周期数} \times \text{时钟周期}\end{aligned}$$

上述公式中，程序所含时钟周期数可由程序所含指令条数和相应的 CPI 求得。

如果已知程序总的指令条数和综合 CPI，则可用如下公式计算程序的总时钟周期数。

$$\text{程序总时钟周期数} = \text{程序所含指令条数} \times \text{CPI}$$

如果已知程序中共有 n 种不同类型的指令，第 i 种指令的条数和 CPI 分别为 C_i 和 CPI_i，则可用如下公式计算程序总时钟周期数。

$$\text{程序总时钟周期数} = \sum_{i=1}^{n} (\text{CPI}_i \times C_i)$$

程序的综合 CPI 也可由以下公式求得，其中，F_i 表示第 i 种指令在程序中所占的比例。

$$\text{CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times F_i) = \text{程序总时钟周期数} \div \text{程序所含指令条数}$$

因此，若已知程序综合 CPI 和指令条数，则可用下列公式计算 CPU 时间。

$$\text{CPU 执行时间} = \text{CPI} \times \text{程序总指令条数} \times \text{时钟周期}$$

有了 CPU 执行时间，就可以评判两台计算机性能的好坏。计算机的性能可以看成是 CPU 时间的倒数，因此，两台计算机性能之比就是 CPU 时间之比的倒数。若计算机 M1 和 M2 的性能之比为 n，则说明“M1 的速度是 M2 的速度的 n 倍”，也就是说，“在 M2 上执行程

序的时间是在 M1 上执行时间的 n 倍”。

CPU 时间度量公式中的时钟周期、指令条数、CPI 三个因素是相互制约的。例如，更改指令集可以减少程序所含指令的条数，但是，同时可能引起 CPU 结构的调整，从而可能会增加时钟周期的宽度(即降低时钟频率)。对于解决同一个问题的不同程序，即使是在同一台计算机上，指令条数最少的程序也不一定执行得最快。有关时钟周期、指令条数和 CPI 的相互制约关系，在学完后面有关指令系统设计、数据通路设计等章节后，会有更深刻的认识和理解。

例 1.1 假设某个频繁使用的程序 P 在机器 M1 上运行需要 10 秒钟，M1 的时钟频率为 2GHz。设计人员想开发一台与 M1 具有相同 ISA 的新机器 M2。采用新技术可使 M2 的时钟频率增加，但同时也会使 CPI 增加。假定程序 P 在 M2 上的时钟周期数是在 M1 上的 1.5 倍，则 M2 的时钟频率至少达到多少才能使程序 P 在 M2 上的运行时间缩短为 6 秒钟？

解：程序 P 在机器 M1 上的时钟周期数为 CPU 执行时间×时钟频率＝10s×2GHz＝20G。因此，程序 P 在机器 M2 上的时钟周期数为 1.5×20G＝30G。要使程序 P 在 M2 上运行时间缩短到 6s，则 M2 的时钟频率至少应为：程序所含时钟周期数÷CPU 执行时间＝30G/6s＝5GHz。由此可见，M2 的时钟频率是 M1 的 2.5 倍，但 M2 的速度却只是 M1 的 1.67 倍。

上述例子说明，由于时钟频率的提高可能会对 CPU 结构带来影响，从而使其他性能指标降低，因此，虽然时钟频率的提高会加快 CPU 执行程序的速度，但不能保证执行速度有同倍数的提高。

例 1.2 假设计算机 M 的指令集中包含 A、B、C 三类指令，其 CPI 分别为 1、2、4。某个程序 P 在 M 上被编译成两个不同的目标代码序列 P1 和 P2，P1 所含 A、B、C 三类指令的条数分别为 8、2、2，P2 所含 A、B、C 三类指令的条数分别为 2、5、3。请问：哪个代码序列指令条数少？哪个执行速度快？它们的 CPI 分别是多少？

解：P1 和 P2 的指令条数分别为 12 和 10，所以，P2 的指令条数少。

P1 的时钟周期数为 8×1＋2×2＋2×4＝20。

P2 的时钟周期数为 2×1＋5×2＋3×4＝24。

因为两个序列在同一台机器上运行，所以时钟周期一样，故时钟数少的代码序列所用时间短、执行速度快。显然，P1 比 P2 快。

从上述结果来看，指令数少的代码序列执行时间并不更短。

CPI＝程序总时钟周期数÷程序所含指令条数，

因此，P1 的 CPI 为 20/12＝1.67；P2 的 CPI 为 24/10＝2.4。

上述例子说明，指令条数少并不代表执行时间短，同样，时钟频率高也不说明执行速度快。在评价计算机性能时，仅考虑单个因素是不全面的，必须三个因素同时考虑。1.6.3 节介绍的性能指标 MIPS 曾被普遍使用，它就没有考虑所有三个因素，所以用它来评价性能有时会得到不准确的结论。

1.6.3 用指令执行速度进行性能评估

最早用来衡量计算机性能的指标是每秒钟完成单个运算(如加法运算)指令的条数。当时大多数指令的执行时间是相同的,并且加法指令能反映乘、除等运算性能,其他指令的时间大体与加法指令相当,故加法指令的速度有一定的代表性。指令速度所用的计量单位为MIPS(Million Instructions Per Second),其含义是平均每秒钟执行多少百万条指令。

早期还有一种类似于MIPS的性能估计方式,就是指令平均执行时间,也称等效指令速度法或Gibson混合法。随着计算机体系结构的发展,不同指令所需的执行时间差别越来越大,人们就根据等效指令速度法通过统计各类指令在程序中所占比例进行折算。设某类指令i在程序中所占比例为w_i,执行时间为t_i,则等效指令的执行时间为$T=w_1\times t_1+w_2\times t_2+\cdots+w_n\times t_n$($n$为指令种类数)。如果指令执行时间用时钟周期数来衡量的话,则上式计算的结果就是CPI。对指令平均执行时间求倒数能够得到MIPS值。

选取一组指令组合,使得得到的平均CPI最小,由此得到的MIPS就是峰值MIPS(Peak MIPS)。有些制造商经常将峰值MIPS直接当作MIPS,而实际上的性能要比标称的性能差。

相对MIPS(Relative MIPS)是根据某个公认的参考机型来定义的相应MIPS值,其值的含义是被测机型相对于参考机型MIPS的多少倍。

MIPS反映的是机器执行定点指令的速度。但是,不同机器的指令集不同,指令的功能也不同,也许在机器M1上某一条指令的功能,在机器M2上要用多条指令来完成,因此,同样的指令条数所完成的功能可能完全不同;另外,不同机器的CPI和时钟周期也不同,因而同一条指令在不同机器上所用的时间也不同。所以,用MIPS来对不同的机器进行性能比较有时是不准确或不客观的,下面的例子可以说明这点。

例1.3 假定某程序P编译后生成的目标代码由A、B、C、D四类指令组成,它们在程序中所占的比例分别为43%、21%、12%、24%,已知它们的CPI分别为1、2、2、2。现重新对程序P进行编译优化,生成的新目标代码中A类指令条数减少了50%,其他类指令的条数没有变。请回答下列问题。

(1) 编译优化前后程序的CPI各是多少?

(2) 假定程序在一台主频为50MHz的计算机上运行,则优化前后的MIPS各是多少?

解:优化后A类指令的条数减少了50%,因而各类指令所占比例分别计算如下。

A类指令:$21.5/(21.5+21+12+24)\times 100\% = 27\%$

B类指令:$21/(21.5+21+12+24)\times 100\% = 27\%$

C类指令:$12/(21.5+21+12+24)\times 100\% = 15\%$

D类指令:$24/(21.5+21+12+24)\times 100\% = 31\%$

(1) 优化前后程序的CPI分别计算如下。

优化前:$43\%\times 1+21\%\times 2+12\%\times 2+24\%\times 2 = 1.57$

优化后:$27\%\times 1+27\%\times 2+15\%\times 2+31\%\times 2 = 1.73$

(2) 优化前后程序的MIPS分别计算如下。

优化前:$50M/1.57 = 31.8$ MIPS

优化后:$50M/1.73 = 28.9$ MIPS

从 MIPS 数来看,优化后程序执行速度反而变慢了。

这显然是错误的,因为优化后只减少了 A 类指令条数而其他指令数没变,所以程序执行时间一定减少了。从这个例子可以看出,用 MIPS 数来进行性能估计是不可靠的。

与定点指令运行速度 MIPS 相对应的用来表示浮点操作速度的指标是 MFLOPS (Million FLOating-point operations Per Second)。它表示每秒所执行的浮点运算有多少百万次,它是基于所完成的操作次数而不是指令数来衡量的。

1.6.4 用基准程序进行性能评估

基准程序(benchmarks)是进行计算机性能评测的一种重要工具。基准程序是专门用来进行性能评价的一组程序,能够很好地反映机器在运行实际负载时的性能,可以通过在不同机器上运行相同的基准程序来比较在不同机器上的运行时间,从而评测其性能。基准程序最好是用户经常使用的一些实际程序,或是某个应用领域的一些典型的简单程序。对于不同的应用场合,应该选择不同的基准程序。例如,对用于软件开发的计算机进行评测时,最好选择包含编译器和文档处理软件的一组基准程序。而如果是对用于 CAD 处理的计算机进行评测时,最好选择一些典型的图形处理小程序作为一组基准程序。

基准程序是一个测试程序集,由一组程序组成。例如,SPEC 测试程序集是应用最广泛、也是最全面的性能评测基准程序集。1988 年,由 Sun, MIPS, HP, Apollo, DEC 五家公司联合提出了 SPEC 标准。它包括一组标准的测试程序、标准输入和测试报告。这些测试程序是一些实际的程序,包括系统调用、I/O 等。最初提出的基准程序集分成两类:整数测试程序集 SPECint 和浮点测试程序集 SPECfp。后来分成了按不同性能测试用的基准程序集,如 CPU 性能测试集(SPEC CPU2000)、Web 服务器性能测试集(SPECweb99)等。

如果基准测试程序集中不同的程序在两台机器上测试得出的结论不同,则如何给出最终的评价结论呢?例如,假定基准测试程序集包含有程序 P1 和 P2,程序 P1 在机器 M1 和机器 M2 上运行的时间分别是 10 秒和 2 秒,程序 P2 在机器 M1 和机器 M2 上运行的时间分别是 120 秒和 600 秒,也即:对于 P1,M2 的速度是 M1 的 5 倍;而对于 P2,M1 的速度是 M2 的 5 倍,那么,到底是 M1 还是 M2 更快呢?

可以用所有程序的执行时间之和来比较,但一般采用执行时间的算术平均值或几何平均值来综合评价机器的性能,根据算术平均值能得到总的平均执行时间,而根据几何平均值则不能得到程序总的执行时间。如果考虑每个程序的使用频度,用加权平均的方式,结果会更准确。

也可以将执行时间进行归一化来得到被测试的机器相对于参考机器的性能。

执行时间的归一化值=参考机器上的执行时间÷被测机器上的执行时间

例如,SPEC 比值(SPEC ratio)是指将测试程序在 Sun SPARCstation 上运行时的执行时间除以该程序在测试机器上的执行时间所得到的比值。比值越大,机器的性能越好。

使用基准程序进行计算机性能评测也存在一些缺陷,因为基准程序的性能可能与某一小段的短代码密切相关,此时,硬件系统设计人员或编译器开发者可能会针对这些代码片段进行特殊的优化,使得执行这段代码的速度非常快,以至于得到不准确的性能评测结果。例如,Intel Pentium 处理器运行 SPECint 时用了公司内部使用的特殊编译器,使其性能表现

得很高，但用户实际使用的是普通编译器，达不到所标称的性能。又如，矩阵乘法程序SPEC matrix 300有99%的时间运行在一行语句上，有些厂商用特殊编译器优化该语句，使性能达到VAX 11/780的729.8倍！

1.7 本章小结

本章主要对计算机系统做了概述性的介绍，指出了本教材内容在整个计算机系统中的位置，并对计算机系统的性能评价做了简要说明。

具体总结如下。

- 冯·诺依曼计算机结构的主要特点
  - 由运算器、控制器、存储器、输入设备和输出设备5大部分组成。
  - 指令和数据用二进制表示，两者形式上没有差别。
  - 指令和数据存放在存储器中，按地址访问。
  - 指令由操作码和地址码组成，操作码指定操作性质，地址码指定操作数地址。
  - 采用“存储程序”方式进行工作。
- 计算机硬件的基本组成和功能
  - 运算器用来进行各种算术逻辑运算。
  - 控制器用来执行指令，送出操作控制信号。
  - 存储器用来存放指令和数据。
  - 输入和输出设备用来实现计算机和用户之间的信息交换。
- 计算机系统分软件和硬件两部分，软件和硬件的界面是指令集体系结构（ISA）。
- 计算机系统层次结构
  - 从高到低粗分为：应用软件、系统软件和硬件三个层次。
  - 从高到低细分为：应用程序级（最终用户）、高级语言虚拟机级（高级语言程序员或应用程序员）、汇编语言虚拟机级（汇编语言程序员）、操作系统虚拟机级（系统管理员）、机器语言机器级（机器语言程序员）。
- 硬件和软件的相互关系
  - 两者相辅相成，缺一不可。
  - 两者都用来实现逻辑功能，同一功能可用硬件实现，也可用软件实现。
- 计算机工作过程
  - 用某种语言（高级语言或低级语言）编制源程序。
  - 用语言处理程序（编译程序或汇编程序）将源程序翻译成机器语言目标程序。
  - 将所含的指令和数据装入内存，然后从第一条指令开始执行。
  - 指令执行过程：取指令、指令译码、取操作数，运算、送结果、PC指向下一条指令。
  - 重复上一步，周而复始地执行指令，直到程序所含指令全部执行完。
- 基本性能指标包括程序的响应时间、系统的吞吐率。
- 处理器的基本性能参数有时钟周期（或主频）、CPI、MIPS、MFLOPS。

- 性能的测量
  - 一般把程序的响应时间划分成 CPU 时间和等待时间，CPU 时间又分成用户 CPU 时间和系统 CPU 时间。
  - 因为操作系统对自己所花费的时间进行测量时，不十分准确，所以，对 CPU 性能的测量一般通过测量程序运行的用户 CPU 时间来进行。
- 各种性能指标之间的关系
  - CPU 执行时间＝CPU 时钟周期数×时钟周期
  - 时钟周期和时钟频率互为倒数
  - CPU 时钟周期数＝程序指令数×程序的 CPI
- 性能评价程序（基准程序）
  - 采用一组基准测试程序来对机器的性能进行评测。
  - 有些机器制造商会针对评测程序中频繁出现的语句采用专门的编译器进行优化，使评测程序的运行效率大幅提高，因此有时基准评测程序也不能说明问题。

习　题　1

1. 给出以下概念的解释说明。

（1）系列机　（2）兼容性　（3）中央处理器（CPU）　（4）算术逻辑单元（ALU）
（5）数据通路　（6）控制器　（7）主存　（8）系统软件
（9）应用软件　（10）高级语言　（11）汇编语言　（12）机器语言
（13）源程序　（14）目标程序　（15）编译程序　（16）解释程序
（17）汇编程序　（18）操作系统　（19）最终用户　（20）系统管理员
（21）应用程序员　（22）系统程序员　（23）指令系统　（24）指令集体系结构（ISA）
（25）透明性　（26）响应时间　（27）吞吐率　（28）CPU 执行时间
（29）用户 CPU 时间　（30）系统 CPU 时间　（31）系统性能　（32）CPU 性能
（33）时钟周期　（34）主频　（35）CPI　（36）基准测试程序
（37）SPEC 基准程序集　（38）SPEC 比值　（39）MIPS　（40）峰值 MIPS
（41）相对 MIPS　（42）MFLOPS

2. 简单回答下列问题。

（1）冯·诺依曼计算机由哪几部分组成？各部分的功能是什么？采用什么工作方式？

（2）摩尔定律的主要内容是什么？

（3）计算机系统的层次结构如何划分？计算机系统的用户可分哪几类？每类用户工作在哪个层次？

（4）程序的 CPI 与哪些因素有关？

（5）为什么说性能指标 MIPS 不能很好地反映计算机的性能？

3. 假定你的朋友不太懂计算机，请用简单通俗的语言给你的朋友介绍计算机系统是如何工作的。

4. 你对计算机系统的哪些部分最熟悉，哪些部分最不熟悉？最想进一步了解细节的是哪些部分的内容？

5. 若有两个基准测试程序 P1 和 P2 在机器 M1 和 M2 上运行，假定 M1 和 M2 的价格分别是 5000 元和 8000 元，表 1.2 给出了 P1 和 P2 在 M1 和 M2 上所用的时间和指令条数。

表 1.2　P1 和 P2 在 M1 和 M2 上所用的时间和指令条数

程序	M1		M2	
	指令条数	执行时间	指令条数	执行时间
P1	200×10^6	10 000ms	150×10^6	5000ms
P2	300×10^3	3ms	420×10^3	6ms

请回答下列问题。

(1) 对于 P1,哪台机器的速度快?快多少?对于 P2 呢?

(2) 在 M1 上执行 P1 和 P2 的速度分别是多少 MIPS?在 M2 上的执行速度又各是多少?从执行速度来看,对于 P2,哪台机器的速度快?快多少?

(3) 假定 M1 和 M2 的时钟频率各是 800MHz 和 1.2GHz,则在 M1 和 M2 上执行 P1 时的平均时钟周期数 CPI 各是多少?

(4) 如果某个用户需要大量使用程序 P1,并且该用户主要关心系统的响应时间而不是吞吐率,那么,该用户需要大批构进机器时,应该选择 M1 还是 M2?为什么?(提示:从性价比上考虑)

(5) 如果另一个用户也需要购进大批机器,但该用户使用 P1 和 P2 一样多,主要关心的也是响应时间,那么,应该选择 M1 还是 M2?为什么?

6. 若机器 M1 和 M2 具有相同的指令集,其时钟频率分别为 1GHz 和 1.5GHz。在指令集中有 5 种不同类型的指令 A~E。表 1.3 给出了在 M1 和 M2 上每类指令的平均时钟周期数 CPI。

表 1.3　在 M1 和 M2 上每类指令的平均时钟周期数 CPI

机器	A	B	C	D	E
M1	1	2	2	3	4
M2	2	2	4	5	6

请回答下列问题。

(1) M1 和 M2 的峰值 MIPS 各是多少?

(2) 假定某程序 P 的指令序列中,5 类指令具有完全相同的指令条数,则程序 P 在 M1 和 M2 上运行时,哪台机器更快?快多少?在 M1 和 M2 上执行程序 P 时的 CPI 各是多少?

7. 假设同一套指令集用不同的方法设计了两种机器 M1 和 M2。机器 M1 的时钟周期为 0.8ns,机器 M2 的时钟周期为 1.2ns。某个程序 P 在机器 M1 上运行时的 CPI 为 4,在 M2 上的 CPI 为 2。对于程序 P 来说,哪台机器的执行速度更快?快多少?

8. 假设某机器 M 的时钟频率为 4GHz,用户程序 P 在 M 上的指令条数为 8×10^9,其 CPI 为 1.25,则 P 在 M 上的执行时间是多少?若在机器 M 上从程序 P 开始启动到执行结束所需的时间是 4s,则 P 占用的 CPU 时间的百分比是多少?

9. 假定某编译器对某段高级语言程序编译生成两种不同的指令序列 S1 和 S2,在时钟频率为 500MHz 的机器 M 上运行,目标指令序列中用到的指令类型有 A、B、C 和 D 四类。四类指令在 M 上的 CPI 和两个指令序列所用的各类指令条数如表 1.4 所示。

表 1.4　A、B、C、D 四类指令在 M 上的 CPI 和两个指令序列所用的各类指令条数

	A	B	C	D
各指令的 CPI	1	2	3	4
S1 的指令条数	5	2	2	1
S2 的指令条数	1	1	1	5

请回答：S1 和 S2 各有多少条指令？CPI 各为多少？所含的时钟周期数各为多少？执行时间各为多少？

10. 假定机器 M 的时钟频率为 1.2GHz，某程序 P 在机器 M 上的执行时间为 12 秒。对 P 优化时，将其所有的乘 4 指令都换成了一条左移两位的指令，得到优化后的程序 P′。已知在 M 上乘法指令的 CPI 为 5，左移指令的 CPI 为 2，P 的执行时间是 P′执行时间的 1.2 倍，则 P 中有多少条乘法指令被替换成了左移指令被执行？

第 2 章 数据的机器级表示

数据是计算机处理的对象。从不同的处理角度来看,数据有不同的表现形态。从外部形式来看,计算机可处理数值、文字、图、声音、视频以及各种模拟信息量。从算法描述的角度来看,有图、表、树、队列、矩阵等结构类型的数据。从高级语言程序员的角度来看,有数组、结构、指针、实数、整数、布尔数、字符和字符串等类型的数据。不管以什么形态出现,在计算机内部数据最终都由机器指令来处理。从计算机指令集体系结构(Instruction Set Architecture,ISA)角度来看,计算机中底层的机器级表示数据只有几类简单的基本数据类型,由它们可以组合成各种复杂类型的数据。

本章重点讨论计算机内部数据的机器级表示方式。主要内容包括进位计数制、二进制定点数的编码表示、无符号整数和带符号整数的表示、IEEE 754 浮点数表示标准、西文字符和汉字的编码表示、十进制数的二进制编码表示(即 BCD 码)、C 语言中各种类型数据的表示和转换、数据的宽度和存放顺序以及几种常用检/纠错码的编码表示与使用方法。

2.1 数制和编码

*2.1.1 信息的二进制编码

计算机内部处理的所有数据都必须是"数字化编码"了的数据。现实世界中的感觉媒体信息(如声音、文字、图画、活动图像等)由输入设备转化为二进制编码表示,因此,输入设备必须具有"离散化"和"编码"两方面的功能。因为计算机中用来存储、加工和传输数据的部件都是位数有限的部件,所以计算机中只能表示和处理离散的信息。"数字化编码"过程,就是指对感觉媒体信息进行定时采样,将现实世界中的连续信息转换为计算机中的离散的"样本"信息,然后对它们用"0"和"1"进行数字化编码的过程。

所谓编码,就是用少量简单的基本符号,对大量复杂多样的信息进行一定规律的组合。基本符号的种类和组合规则是信息编码的两大要素。例如,电报码中用 4 位十进制数字表示汉字;从键盘上输入汉字时用汉语拼音(即 26 个英文字母)表示汉字等,都是编码的典型例子。

在计算机系统内部,所有信息都是用二进制进行编码的。也就是说计算机内部采用的是二进制表示方式。这样做的原因有以下几点。

(1) 二进制只有两种基本状态,使用有两个稳定状态的物理器件就可以表示二进制数的每一位,而制造有两个稳定状态的物理器件要比制造有多个稳定状态的物理器件容易得

多。例如用高、低两个电位,或用脉冲的有无,或脉冲的正负极性等都可以很方便、很可靠地表示“0”和“1”。

(2) 二进制的编码、计数和运算规则都很简单。可用开关电路实现,简便易行。

(3) 两个符号1和0正好与逻辑命题的两个值“真”和“假”相对应,为计算机中实现逻辑运算和程序中的逻辑判断提供了便利的条件,特别是能通过逻辑门电路方便地实现算术运算。

采用二进制编码将各种媒体信息转变成数字化信息后,可以在计算机内部进行存储、处理和传送。在高级语言程序中,可以用图、树、链表和队列等进行算法描述,并能以数组、结构、指针和字符串等数据类型来说明处理对象,但将高级语言程序转换为机器语言程序后,每条指令的操作数就只能是某种简单的基本数据类型。

如图2.1中虚线框所示,指令所处理的基本数据类型分为两种:数值型数据和非数值型数据。数值型数据可用来表示数量的多少,可比较其大小,分为整数和实数,整数又分为无符号整数和带符号整数。在计算机内部,整数用定点数表示,实数用浮点数表示。非数值型数据没有大小之分,不表示数量的多少,主要包括字符数据和逻辑数据。

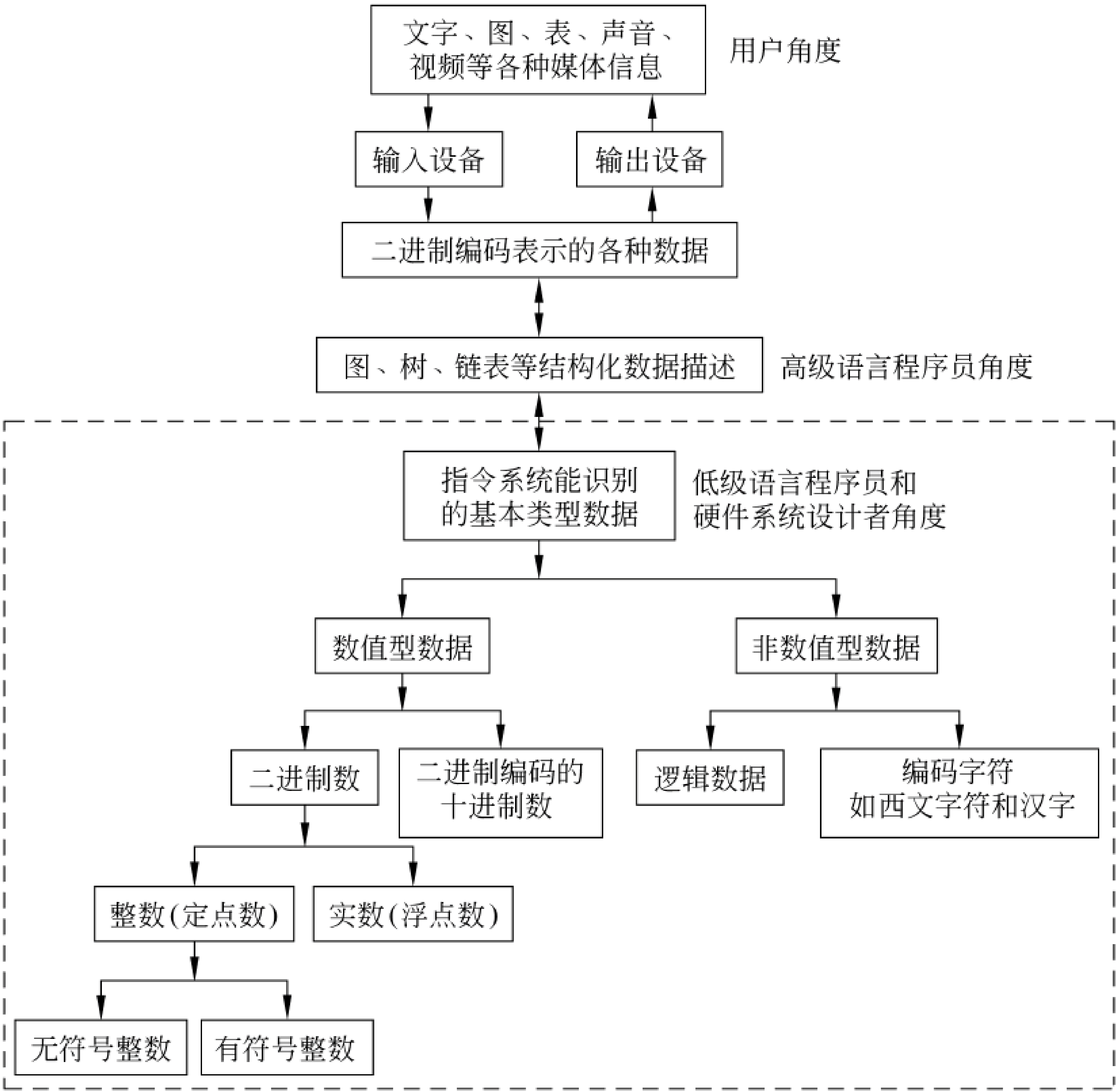


图2.1 计算机外部信息与内部数据的转换

日常生活中,常使用带正负号的十进制数表示数值数据,例如6.18,−127等。但这种形式的数据在计算机内部难以直接存储、运算和传输。通常的十进制数仅仅是一种数值数据的输入输出形式,而不是计算机内部的表示形式。

在计算机内部，数值数据的表示方法有两种，第一种是直接用二进制数表示；另一种是采用二进制编码的十进制数(Binary Coded Decimal Number，BCD)表示。

表示一个数值数据要确定三个要素：进位计数制、定/浮点表示和编码规则。任何给定的一个二进制0/1序列，在未确定它采用什么进位计数制、定点还是浮点表示以及编码表示方法之前，它所代表的数值数据的值是无法确定的。

*2.1.2 进位计数制

日常生活中基本上都使用十进制数，其每个数位可用10个不同符号0，1，2，…，9来表示，每个符号处在十进制数中不同位置时，所代表的数值不一样。例如，2585.62代表的值是：

$$(2585.62)_{10}=2\times10^3+5\times10^2+8\times10^1+5\times10^0+6\times10^{-1}+2\times10^{-2}$$

一般地，任意一个十进制数

$$D=d_nd_{n-1}\cdots d_1d_0.d_{-1}d_{-2}\cdots d_{-m}\quad(m,n\text{ 为正整数}),$$

其值可表示成如下形式。

$$\begin{aligned}V(D)=&d_n\times10^n+d_{n-1}\times10^{n-1}+\cdots+d_1\times10^1+d_0\times10^0\\&+d_{-1}\times10^{-1}+d_{-2}\times10^{-2}+\cdots+d_{-m}\times10^{-m}\end{aligned}$$

其中$d_i(i=n,n-1,\cdots,1,0,-1,-2,\cdots,-m)$可以是0，1，2，3，4，5，6，7，8，9这10个数字符号中的任何一个，“10”称为基数(base)，它代表每个数位上可以使用的不同数字符号个数。10^i称为第i位上的权。在十进制数进行运算时，每位计满10之后就要向高位进一，即日常所说的“逢十进一”。

类似地，二进制数的基数是2，只使用两个不同的数字符号0和1，运算时采用“逢二进一”的规则，第i位上的权是2^i。例如，二进制数$(100101.01)_2$代表的值是：

$$\begin{aligned}(100101.01)_2&=1\times2^5+0\times2^4+0\times2^3+1\times2^2+0\times2^1+1\times2^0+0\times2^{-1}+1\times2^{-2}\\&=(37.25)_{10}\end{aligned}$$

一般地，任意一个二进制数

$$B=b_nb_{n-1}\cdots b_1b_0.b_{-1}b_{-2}\cdots b_{-m}\quad(m,n\text{ 为正整数}),$$

其值可表示为如下形式。

$$\begin{aligned}V(B)=&b_n\times2^n+b_{n-1}\times2^{n-1}+\cdots+b_1\times2^1+b_0\times2^0+b_{-1}\times2^{-1}\\&+b_{-2}\times2^{-2}+\cdots+b_{-m}\times2^{-m}\end{aligned}$$

其中$b_i(i=n,n-1,\cdots,1,0,-1,-2,\cdots,-m)$只可以是0和1两种不同的数字符号。

扩展到一般情况，在R进制数字系统中，应采用R个基本符号$(0,1,2,\cdots,R-1)$表示各位上的数字，采用“逢R进一”的运算规则，对于每一个数位i，该位上的权为R^i。R被称为该数字系统的基。

在计算机系统中，常用的几种进位计数制有下列几种。

二进制$R=2$，基本符号为0和1。

八进制$R=8$，基本符号为0，1，2，3，4，5，6，7。

十六进制$R=16$，基本符号为0，1，2，3，4，5，6，7，8，9，A，B，C，D，E，F。

十进制$R=10$，基本符号为0，1，2，3，4，5，6，7，8，9。

表2.1列出了二、八、十、十六进制4种进位计数制中各基本数之间的对应关系。

表 2.1　4 种进制数之间的对应关系

二进制数	八进制数	十进制数	十六进制数
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

从表 2.1 中可看出，十六进制的前 10 个数字与十进制中前 10 个数字相同，后 6 个基本符号 A，B，C，D，E，F 的值分别为十进制的 10，11，12，13，14，15。在书写时可使用后缀字母标识该数的进位计数制，一般用 B（Binary）表示二进制，用 O（Octal）表示八进制，用 D（Decimal）表示十进制（十进制数的后缀可以省略），而 H（Hexadecimal）则是十六进制数的后缀，例如二进制数 10011B，十进制数 56D 或 56，十六进制数 308FH，3C.5H 等。

计算机内部所有的信息采用二进制编码表示。但在计算机外部，为了书写和阅读的方便，大都采用八、十或十六进制表示形式。因此，计算机在数据输入后或输出前都必须实现这些进位制数和二进制数之间的转换。以下介绍各进位计数制之间数据的转换方法。

1. *R* 进制数转换成十进制数

任何一个 *R* 进制数转换成十进制数时，只要“按权展开”即可。

例 2.1　将二进制数 $(10101.01)_2$ 转换成十进制数。

解：$(10101.01)_2=(1\times2^4+0\times2^3+1\times2^2+0\times2^1+1\times2^0+0\times2^{-1}+1\times2^{-2})_{10}$

$=(21.25)_{10}$。

例 2.2　将八进制数 $(307.6)_8$ 转换成十进制数。

解：$(307.6)_8=(3\times8^2+7\times8^0+6\times8^{-1})_{10}=(199.75)_{10}$。

例 2.3　将十六进制数 $(3A.C)_{16}$ 转换成十进制数。

解：$(3A.C)_{16}=(3\times16^1+10\times16^0+12\times16^{-1})_{10}=(58.75)_{10}$。

2. 十进制数转换成 R 进制数

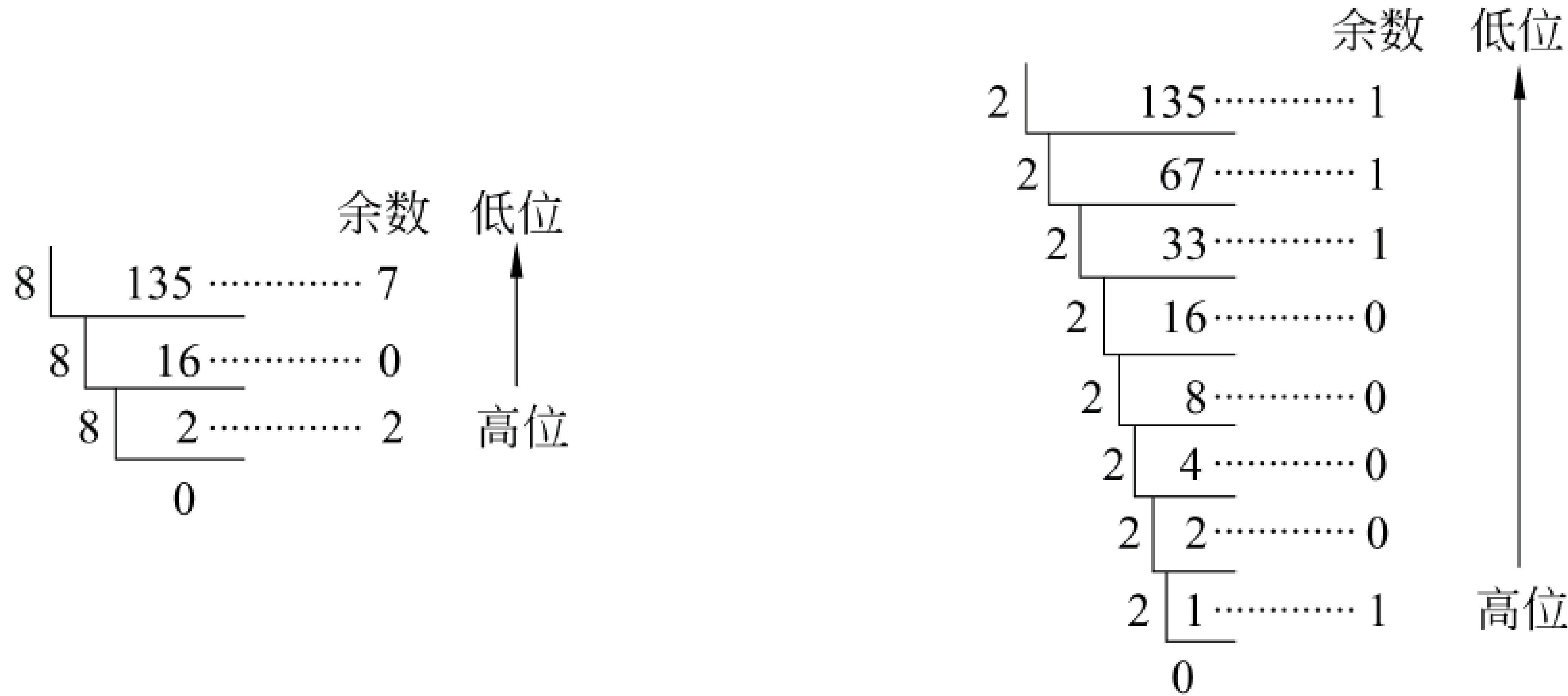
任何一个十进制数转换成 R 进制数时，要将整数和小数部分分别进行转换。

1）整数部分的转换

整数部分的转换方法是“除基取余，上右下左”。也就是说，用要转换的十进制整数去除以基数 R，将得到的余数作为结果数据中各位的数字，直到余数为 0 为止。上面的余数（即先得到的余数）作为右边低位上的数位，下面的余数作为左边高位上的数位。

例 2.4 将十进制整数 135 分别转换成八进制数和二进制数。

解：将 135 分别除以 8 和 2，将每次的余数按从低位到高位的顺序排列如下。



所以，$(135)_{10}=(207)_8=(10000111)_2$。

2）小数部分的转换

小数部分的转换方法是“乘基取整，上左下右”。也就是说，用要转换的十进制小数去乘以基数 R，将得到的乘积的整数部分作为结果数据中各位的数字，小数部分继续与基数 R 相乘。以此类推，直到某一步乘积的小数部分为 0 或已得到希望的位数为止。最后，将上面的整数部分作为左边高位上的数位，下面的整数部分作为右边低位上的数位。

例 2.5 将十进制小数 0.6875 分别转换成二进制数和八进制数。

解：$0.6875\times2=1.375$　　整数部分=1　（高位）

$0.375\times2=0.75$　　整数部分=0　↓

$0.75\times2=1.5$　　整数部分=1　↓

$0.5\times2=1.0$　　整数部分=1　（低位）

所以，$(0.6875)_{10}=(0.1011)_2$。

$0.6875\times8=5.5$　　整数部分=5　（高位）

$0.5\times8=4.0$　　整数部分=4　（低位）

所以，$(0.6875)_{10}=(0.54)_8$。

在转换过程中，可能乘积的小数部分总得不到 0，即转换得到希望的位数后还有余数，这种情况下得到的是近似值，如下例所示。

例 2.6 将十进制小数 0.63 转换成二进制数。

解：$0.63\times2=1.26$　　整数部分=1　（高位）

$0.26\times2=0.52$　　整数部分=0　↓

$0.52\times2=1.04$　　整数部分=1　↓

$0.04\times2=0.08$　　整数部分=0　（低位）

所以，$(0.63)_{10}=(0.1010\cdots)_2$。

3）含整数、小数部分的数的转换

只要将整数部分和小数部分分别进行转换，得到转换后相应的整数和小数部分，然后再将这两部分组合起来得到一个完整的数。

例 2.7 将十进制数 135.6875 分别转换成二进制数和八进制数。

解：$(135.6875)_{10}=(10000111.1011)_2=(207.54)_8$。

3. 二进制、八进制、十六进制数的相互转换

1）八进制数转换成二进制数

八进制数转换成二进制数的方法很简单，只要把每一个八进制数字改写成等值的 3 位二进制数即可，且保持高低位的次序不变。八进制数字与二进制数的对应关系如下。

$(0)_8=000$ $(1)_8=001$ $(2)_8=010$ $(3)_8=011$

$(4)_8=100$ $(5)_8=101$ $(6)_8=110$ $(7)_8=111$

例 2.8 将$(13.724)_8$转换成二进制数。

解：$(13.724)_8=(001\ 011.111\ 010\ 100)_2=(1011.1110101)_2$。

2）十六进制数转换成二进制数

十六进制数转换成二进制数的方法与八进制数转换成二进制数的方法类似，只要把每一个十六进制数字改写成等值的 4 位二进制数即可，且保持高低位的次序不变。十六进制数字与二进制数的对应关系如下。

$(0)_{16}=0000$ $(1)_{16}=0001$ $(2)_{16}=0010$ $(3)_{16}=0011$

$(4)_{16}=0100$ $(5)_{16}=0101$ $(6)_{16}=0110$ $(7)_{16}=0111$

$(8)_{16}=1000$ $(9)_{16}=1001$ $(A)_{16}=1010$ $(B)_{16}=1011$

$(C)_{16}=1100$ $(D)_{16}=1101$ $(E)_{16}=1110$ $(F)_{16}=1111$

例 2.9 将十六进制数$(2B.5E)_{16}$转换成二进制数。

解：$(2B.5E)_{16}=(0010\ 1011.0101\ 1110)_2=(101011.0101111)_2$。

3）二进制数转换成八进制数

二进制数转换成八进制数时，整数部分从低位向高位方向每 3 位用一个等值的八进制数来替换，最后不足 3 位时在高位补 0 凑满 3 位；小数部分从高位向低位方向每 3 位用一个等值的八进制数来替换，最后不足 3 位时在低位补 0 凑满 3 位。例如，

$$(0.10101)_2=(000.101\ 010)_2=(0.52)_8$$

$$(10011.01)_2=(010\ 011.010)_2=(23.2)_8$$

4）二进制数转换成十六进制数

二进制数转换成十六进制数时，整数部分从低位向高位方向每 4 位用一个等值的十六进制数来替换，最后不足 4 位时在高位补 0 凑满 4 位；小数部分从高位向低位方向每 4 位用一个等值的十六进制数来替换，最后不足 4 位时在低位补 0 凑满 4 位。例如，

$$(11001.11)_2=(0001\ 1001.1100)_2=(19.C)_{16}$$

从以上可以看出，二进制数与八进制数、二进制数与十六进制数之间有很简单直观的对应关系。二进制数太长，书写、阅读均不方便；八进制数和十六进制数却像十进制数一样简练，易写易记。虽然计算机中只使用二进制一种计数制。但为了开发和调试程序、阅读机器

代码时的方便，人们经常使用八进制或十六进制来等价地表示二进制，所以大家也必须熟练掌握八进制和十六进制数的表示及其与二进制数之间的转换方法。

2.1.3 定点与浮点表示

日常生活中所使用的数有整数和实数之分，整数的小数点固定在数的最右边，可以省略不写，而实数的小数点则不固定。在计算机中只能识别和表示“0”和“1”，而无法识别小数点，因此，要使得计算机能够处理日常使用的数值数据，必须要解决小数点的表示问题。通常计算机中通过约定小数点的位置来实现。小数点位置约定在固定位置的数称为定点数，小数点位置约定为可浮动的数称为浮点数。

1. 定点表示

定点表示法用来对定点小数和定点整数进行表示。对于定点小数，其小数点总是固定在数的最左边，一般用来表示浮点数的尾数部分。对于定点整数，其小数点总是固定在数的最右边，因此可用“定点整数”来表示整数。

2. 浮点表示

对于任意一个二进制数 X，可以表示成如下形式：

$$X=(-1)^S\times M\times R^E$$

其中 S 取值为 0 或 1，用来决定数 X 的符号；M 是一个二进制定点小数，称为数 X 的尾数(mantissa)；E 是一个二进制定点整数，称为数 X 的阶码或指数(exponent)；R 是基数(radix base)，可以取值为 2、4、16 等。在基数 R 一定的情况下，尾数 M 的位数反映数 X 的有效位数，它决定了数的表示精度，有效位数越多，表示精度就越高；阶码 E 的位数决定数 X 的表示范围；阶码 E 的值确定了小数点的位置。

从浮点数的形式来看，绝对值最小的非零数是如下形式的数：$0.0\cdots01\times R^{-11\cdots1}$，而绝对值最大的数的形式应为 $0.11\cdots1\times R^{11\cdots1}$，所以，假设 m 和 n 分别表示阶码和尾数的位数，基数为 2，则浮点数 X 的绝对值的范围为：

$$2^{-(2^m-1)}\times 2^{-n}\leqslant |X| \leqslant (1-2^{-n})\times 2^{(2^m-1)}$$

上述公式中，紧靠$|X|$左右两边的两个因子就是非零定点小数的绝对值表示范围，浮点数的最小数是定点小数的最小数 2^{-n} 去除以一个很大的数 $2^{(2^m-1)}$，而浮点数的最大数则是定点小数的最大数$(1-2^{-n})$去乘以这个大数 $2^{(2^m-1)}$，由此可见，浮点数的表示范围比定点数要大得多。

2.1.4 定点数的编码表示

定/浮点表示解决了小数点的表示问题。但是，对于一个数值数据来说，还有一个正/负号的表示问题。计算机中只能识别和表示 0 和 1。因此，正/负号也要用 0 和 1 来表示。这种将数的符号用 0、1 表示的处理方式称为符号数字化。一般规定 0 表示正号，1 表示负号。

数字化了的符号能否和数值部分一起参加运算呢？为了解决这个问题，就产生了把符号位和数值部分一起进行编码的各种方法。因为任意一个浮点数都可以用一个定点小数和一个定点整数来表示，所以，只需要考虑定点数的编码表示。

主要有 4 种定点数编码表示方法：原码、补码、反码和移码。通常将数值数据在计算机内部编码表示的数称为机器数，而机器数真正的值(即现实世界中带有正负号的数)称为机

器数的真值。

假设机器数 X 的真值 X_T 为：

$$X_T=\pm X'_{n-1}X'_{n-2}\cdots X'_1X'_0\ (\text{当 } X \text{ 为定点整数时})$$

$$X_T=\pm 0.X'_{n-1}X'_{n-2}\cdots X'_1X'_0\ (\text{当 } X \text{ 为定点小数时})$$

对 X_T用 $n+1$ 位二进制数编码后，机器数 X 表示为：

$$X=X_n\ X_{n-1}\ X_{n-2}\cdots\ X_1\ X_0$$

机器数 X 的位数为 $n+1$，其中，第一位 X_n是数的符号位，后面 n 位 $X_{n-1}\cdots X_1\ X_0$是数值部分。数值数据在计算机内部的编码问题，实际上就是机器数 X 的各位 X_i的取值与真值 X_T 的关系问题。

在上述对机器数 X 和真值 X_T的假设条件下，下面介绍各种定点数的编码表示。

1. 原码表示法

一个数的原码表示由符号位直接跟数值位构成，因此，也称“符号-数值(sign and magnitude)”表示法。原码表示法中，正数和负数的编码表示仅符号位不同，数值部分完全相同。

原码编码规则如下。

(1) 当 X_T为正数时，$X_n=0, X_i=X'_i\ (0\leqslant i\leqslant n-1)$。

(2) 当 X_T为负数时，$X_n=1, X_i=X'_i\ (0\leqslant i\leqslant n-1)$。

原码 0 有两种表示形式：$[+0]_{原}=0\ 00\cdots0$

$$[-0]_{原}=1\ 00\cdots0$$

原码表示的优点是与真值的对应关系直观、方便，因此与真值的转换简单，并且用原码实现乘除运算比较简便。其缺点是 0 的表示不唯一，给使用带来不便。更重要的是，原码加减运算规则复杂，在进行原码加减运算过程中，要判定是否是两个异号数相加或两个同号数相减，若是，则必须判定两个数的绝对值大小，根据判断结果决定结果符号，并用绝对值大的数减去绝对值小的数。现代计算机中不用原码来表示整数，只用定点原码小数来表示浮点数的尾数部分。

2. 补码表示法

补码表示可以实现加减运算的统一，即用加法来实现减法运算。补码表示法也称“2—补码(two's complement)”表示法，由符号位后跟上真值的模 2^n补码构成。因此在介绍补码概念之前，先讲一下有关模运算的概念。

1) 模运算

在模运算系统中，若 A,B,M 满足下列关系：$A=B+K\times M$ (K 为整数)，则记为 $A\equiv B\ (\mathrm{mod}\ M)$。即 A、B 各除以 M 后的余数相同，故称 B 和 A 为模 M 同余。也就是说在一个模运算系统中，一个数与它除以“模”后得到的余数是等价的。

“钟表”是一个典型的模运算系统，其模数为 12。假定现在钟表时针指向 10 点，要将它拨向 6 点，则有两种拨法。

① 倒拨 4 格：$10-4=6$

② 顺拨 8 格：$10+8=18\equiv6\ (\mathrm{mod}\ 12)$

所以在模 12 系统中，$10-4\equiv10+(12-4)\equiv10+8\ (\mathrm{mod}\ 12)$。即 $-4\equiv8\ (\mathrm{mod}\ 12)$。

我们称 8 是 -4 对模 12 的补码。同样有 $-3\equiv9(\mathrm{mod}\ 12)$；$-5\equiv7(\mathrm{mod}\ 12)$等。

由上述例子与同余的概念，可得出如下的结论。

“对于某一确定的模，某数 A 减去小于模的另一数 B，可以用 A 加上 $-B$ 的补码来代替”。这就是为什么补码可以借助加法运算实现减法运算的道理。

例 2.10　假定在“钟表”上只能顺拨时针，则如何用顺拨的方式实现将 10 点倒拨 4 格，拨动后钟表上是几点？

解：“钟表”是一个模运算系统，其模为 12，根据上述结论，可得

$$10-4\equiv 10+(12-4)\equiv 10+8\equiv 6 \pmod{12}$$

因此，可从 10 点顺拨 8（-4 的补码）格，最后是 6 点。

例 2.11　假定算盘只有 4 档，且只能做加法，则如何用该算盘计算 $9828-1928$ 的结果？

解：这个算盘是一个“4 位十进制数”模运算系统，其模为 10^4，根据上述结论，可得

$$9828-1928\equiv 9828+(10^4-1928)\equiv 9828+8072\equiv 7900 \pmod{10^4}$$

因此，用 9828 加 8072（-1928 的补码）即可。

显然，在只有 4 档的算盘上运算时，如果运算结果超过 4 位，则高位无法在算盘上表示，只能用低 4 位表示结果，留在算盘上的值相当于是除以 10^4 后的余数。推广到计算机内部，n 位运算部件就相当于只有 n 档的二进制算盘，其模就是 2^n。

计算机中的存储、运算和传送部件都只有有限位，相当于有限档数的算盘，因此计算机中所表示的机器数的位数也只有有限位。两个 n 位二进制数在进行运算过程中，可能会产生一个多于 n 位的数。此时，计算机和算盘一样，也只能舍弃高位而保留低 n 位，这样做可能会产生两种结果。

① 剩下的低 n 位数不能正确表示运算结果，也即丢掉的高位是运算结果的一部分。例如，在两个同号数相加时，当相加得到的和超出了 n 位数可表示的范围时出现这种情况，我们称此时发生了“溢出（overflow）”现象。

② 剩下的低 n 位数能正确表示运算结果，也即高位的舍去并不影响其运算结果。在两个同号数相减或两个异号数相加时，运算结果就是这种情况。舍去高位的操作相当于“将一个多于 n 位的数去除以 2^n，保留其余数作为结果”的操作，也就是“模运算”操作。

2）补码的定义

根据上述同余概念和数的互补关系，可引出补码的表示：正数的补码是它本身；负数的补码等于模与该负数绝对值之差。因此，数 X_T 的补码可用如下公式表示。

① 当 X_T 为正数时，$[X_T]_补=X_T=M+X_T \pmod{M}$

② 当 X_T 为负数时，$[X_T]_补=M-|X_T|=M+X_T \pmod{M}$

综合①和②，得到结论：对于任意一个数 X_T，$[X_T]_补=M+X_T \pmod{M}$

对于具有一位符号位和 n 位数值位的 $n+1$ 位二进制补码来说，其补码表示定义如下。

① 定点整数：$[X_T]_补=2^{n+1}+X_T \quad (-2^n\leqslant X_T<2^n, \bmod 2^{n+1})$

② 定点小数：$[X_T]_补=2+X_T \quad (-1\leqslant X_T<1, \bmod 2)$

3）特殊数据的补码表示

通过以下例子来说明几个特殊数据的补码表示。

例 2.12　分别求出补码的位数为 n 和 $n+1$ 时“-2^{n-1}”的补码表示。

解：当补码的位数为 n 位时，其模为 2^n。因此：

$[-2^{n-1}]_{补}=2^n-2^{n-1}=2^{n-1}=1\ 0\cdots0$　($n-1$ 个 0)　(mod 2^n)

当补码的位数为 $n+1$ 位时,其模为 2^{n+1}。因此:

$[-2^{n-1}]_{补}=2^{n+1}-2^{n-1}=2^n+2^{n-1}=1\ 10\cdots0$　($n-1$ 个 0)　(mod 2^{n+1})

从该例可以看出,同一个真值在不同位数的补码表示中,其对应的机器数不同。因此,在给定编码表示时,一定要明确编码的位数。在机器内部编码的位数就是机器中运算部件的位数。

例 2.13　设补码的位数为 n,求"-1"的补码表示。

解:对于整数补码有:$[-1]_{补}=2^n-1=11\cdots1$　(n 个 1)

对于小数补码有:$[-1]_{补}=2-1=1.00\cdots0$　($n-1$ 个 0)

由此可见,"-1"既可在整数范围内表示,也能在小数范围内表示,在计算机中有两种不同的补码表示。每个机器数的小数点位置在约定好的默认位置处,也即,上述"-1"的补码小数表示中的小数点"."在计算机内实际上是不存在的,只是书写时为了将小数编码和整数编码区别开来而加上去的。因此,当补码的位数为 n 位时,"-1"的补码小数表示与"-2^{n-1}"的补码表示结果在机器中相同,都是符号位为 1,数值部分为 $n-1$ 个 0。"-1"与"-2^{n-1}"分别是 n 位补码小数和 n 位补码整数中可表示的最小负数。

那么,对于 n 位补码表示来说,2^{n-1} 的补码为多少呢?根据补码定义,得$[2^{n-1}]_{补}=2^n+2^{n-1}$(mod 2^n)$=2^{n-1}=1\ 0\cdots0$,最高位为 1,说明对应的真值应该是负数,这与实际情况不符,因此,显然 n 位补码无法表示 2^{n-1}。由此可以知道,为什么在 n 位补码定义中,真值的取值范围包含了-2^{n-1},但不包含 2^{n-1}。

例 2.14　求 0 的补码表示。

解:根据补码的定义,有

定点小数:$[+0]_{补}=[-0]_{补}=2\pm0=1\ 00\cdots0=0\ 0\cdots0$　(mod 2)

定点整数:$[+0]_{补}=[-0]_{补}=2^{n+1}\pm0=1\ 00\cdots0=0\ 0\cdots0$　(mod 2^{n+1})

从上述结果可知,补码 0 的表示是唯一的。这带来了以下两个方面的好处:

① 减少了$+0$ 和-0 之间的转换。

② 少占用一个编码表示,使补码比原码能多表示一个最小负数。在 n 位原码定点数中,$100\cdots0$ 用来表示-0,但在 n 位补码表示中,-0 和$+0$ 都用全 0 表示,所以可用 $100\cdots0$ 来表示小数中的最小负数-1 或整数中的最小负数-2^{n-1}。

4) 补码与真值之间的转换方法

原码与真值之间的对应关系简单,只要对符号转换,数值部分不需改变。但对于补码来说,正数和负数的转换则不同。根据定义,求一个正数的补码时,只要将正号"+"转换为 0,数值部分无须改变;求一个负数的补码时,需要做减法运算,因而不太方便和直观。

例 2.15　设补码的位数为 8,求 1101100 和-1101100 的补码表示。

解:补码的位数为 8,说明补码数值部分有 7 位,故

$[1101100]_{补}=2^8+1101100=100000000+1101100=01101100$　(mod 2^8)

$[-1101100]_{补}=2^8-1101100=100000000-1101100$

$=10000000+10000000-1101100$

$=10000000+(1111111-1101100)+1$

$=10000000+0010011+1=10010100$　(mod 2^8)

本例中是两个绝对值相同、符号相反的数。其中,负数的补码计算过程中第一个

10000000 用于产生最后的符号 1，第二个 10000000 拆为 1111111＋1，而(1111111－1101100)实际是将数值各位取反。模仿这个计算过程，不难从补码的定义推导出负数补码计算的一般步骤为：符号位为 1，对数值部分“各位取反，末尾加 1”。由定义可以证明，这个规则同样适用于负的定点小数补码的计算。

因此，可以用以下简单方法求一个数的补码：对于正数，符号位取 0，其余同真值中相应各位；对于负数，符号位取 1，其余各位由真值的数值部分“各位取反，末尾加 1”得到。

例 2.16 假定补码位数为 8，用简便方法求数 $X_1=-0.1100011$ 和 $X_2=-1100011$ 的补码表示。

解：$[X_1]_补=1.0011100+0.0000001=1.0011101$。

$[X_2]_补=1\ 0011100+0\ 0000001=1\ 0011101$。

反过来由补码求真值的简便方法为：若符号位为 0，则真值的符号为正，其数值部分不变；若符号位为 1，则真值的符号为负，其数值部分的各位由补码“各位取反，末尾加 1”所得。因为，由以上的负数补码计算方法可以直接想到的方法是，对补码数值部分先减 1 然后再取反。也就是说，通过计算 1111111－(0011101－1)得到，该计算可以变为(1111111－0011101)＋1，亦即进行“取反加 1”操作。从补码定义也不难推导出这个结论。

例 2.17 已知：$[X_T]_补=1\ 0110100$，求真值 X_T。

解：$X_T=-(1001011+1)=-1001100$。

根据上述有关补码和真值转换规则，不难发现，根据补码$[X_T]_补$求 $[-X_T]_补$的方法是，对$[X_T]_补$“各位取反，末尾加 1”。这里要注意最小负数取负后会发生溢出。即最小负数取负后的补码表示是不存在的。

例 2.18 已知：$[X_T]_补=1\ 0110100$，求$[-X_T]_补$。

解：$[-X_T]_补=0\ 1001011+0\ 0000001=0\ 1001100$。

例 2.19 已知：$[X_T]_补=1\ 0000000$，求$[-X_T]_补$。

解：$[-X_T]_补=0\ 1111111+0\ 0000001=1\ 0000000$ (结果溢出)。

例 2.19 中出现了“两个正数相加，结果为负数”的情况，因此，结果是一个错误的值，我们称结果“溢出”，该例中，8 位整数补码 10000000 对应的是最小负数 -2^7，对其取负后的值为 2^7，而 8 位整数补码能表示的最大正数为 2^7-1，因而数太大无法表示，结果溢出。

5) 变形补码

为了便于判断运算结果是否溢出，某些计算机中还采用了一种双符号位的补码表示方式，称为变形补码，因为这种补码小数的模为 4，因此也称为模 4 补码。在双符号位中，左符是真正的符号位，右符用来判别“溢出”。

假定变形补码的位数为 $n+2$(其中符号占两位，数值部分占 n 位)，则变形补码可如下表示。

① 定点整数：$[X_T]_补=2^{n+2}+X_T \quad (-2^n \leqslant X_T < 2^n, \bmod 2^{n+2})$

② 定点小数：$[X_T]_补=4+X_T \quad (-1 \leqslant X_T < 1, \bmod 4)$

例 2.20 已知：$X_T=-1011$，分别求出变形补码取 6 位和 8 位时$[X_T]_{变补}$的值。

解：$[X_T]_{变补}=2^6-1011=100\ 0000-00\ 1011=11\ 0101$。

$[X_T]_{变补}=2^8-1011=100\ 000000-00\ 001011=11\ 110101$。

例 2.21 已知：$X_T=-0.1011$，并假定变形补码取 8 位，求$[X_T]_{变补}$。

解：$[X_T]_{变补}=4-0.1011=100.000000-0.101100=11.010100$。

3. 反码表示法

负数的补码可采用“各位求反，末尾加 1”的方法得到，如果仅各位求反而末尾不加 1，那么就可得到负数的反码表示，因此负数反码的定义就是在相应的补码表示中再末尾减 1。

反码表示存在以下几个方面的不足：0 的表示不唯一；表示范围比补码少一个最小负数；运算时必须考虑循环进位。因此，反码在计算机中很少被使用，有时用作数码变换的中间表示形式。

4. 移码表示法

用浮点数表示一个数值数据时，实际上是用两个定点数来表示的。用一个定点小数表示浮点数的尾数，用一个定点整数表示浮点数的阶码。一般情况下，浮点数的阶码都用一种称之为“移码”的编码方式表示。

为什么要用移码表示阶码呢？因为阶码 E 可以是正数，也可以是负数，当进行浮点数的加减运算时，必须先“对阶”（即比较两个数阶码的大小并使之相等）。为简化比较操作，使操作过程不涉及阶码的符号，可以对每个阶码都加上一个正的常数，称为偏置常数（bias），使所有阶码都转换为正整数，这样，在对浮点数的阶码进行比较时，就是对两个正整数进行比较，因而可以直观地将两个数按位从左到右进行比对，简化了“对阶”操作。

移码的定义如下。

设 E 为阶码，阶码的移码表示位数为 n，则$[E]_{移}=2^{n-1}+E$　（2^{n-1}为偏置常数）。

移码主要用来表示浮点数的阶码，因此，移码只用来表示定点整数。对于 n 位移码$[E]_{移}$，可以得到如下几个结论。

（1）E 的范围为$-2^{n-1}\leqslant E\leqslant 2^{n-1}-1$。

（2）移码 0 的真值为-2^{n-1}，即$[-2^{n-1}]_{移}=00\cdots0$。

（3）0 的移码表示是唯一的，即$[+0]_{移}=[-0]_{移}=2^{n-1}=10\cdots0$（$n-1$ 个 0）。

（4）若将移码第一位看成是符号位，则同一个真值的移码和补码仅符号位不同。这一点不难从补码和移码的定义中看出：两者相差 2^{n-1}，因而符号位相反。

2.2　整数的表示

整数的小数点隐含在数的最右边，故无须表示小数点，因而也被称为定点数。计算机中处理的整数可以用二进制表示，也可以用二进制编码的十进制数（BCD 码）表示。二进制整数分为无符号整数（unsigned integer）和带符号整数（signed integer）两种。

2.2.1　无符号整数的表示

当一个编码的所有二进位都用来表示数值而没有符号位时，该编码表示的就是无符号整数。此时，默认数的符号为正，所以无符号整数就是正整数或非负整数。

一般在全部是正数运算且不出现负值结果的场合下，使用无符号整数表示。例如，可用无符号整数进行地址运算，或用来表示指针。通常把无符号整数简单地说成无符号数。

由于无符号整数省略了一位符号位，所以在字长相同的情况下，它能表示的最大数比带

符号整数所能表示的大，n 位无符号整数可表示的数的范围为 $0\sim(2^n-1)$。例如，8 位无符号整数的形式为 00000000B ～11111111B，对应的数的取值范围为 $0\sim(2^8-1)$，即最大数为 255，而 8 位带符号整数的最大数是 127。

2.2.2 带符号整数的表示

带符号整数也被称为有符号整数，它必须用一个二进位来表示符号，虽然前面介绍的各种二进制定点数编码表示(包括原码、补码、反码和移码)都可以用来表示带符号整数，但是补码表示有其突出的优点。

根据前面对原码、补码、反码和移码等各种定点数编码表示的介绍，不难看出，补码表示具有以下多个方面的优点。

(1) 与原码和反码相比，数 0 的补码表示形式唯一；

(2) 与原码和移码相比，补码运算系统是一种模运算系统，因而可用加法实现减法运算，且符号位可以和数值位一起参加运算；

(3) 与原码和反码相比，它比原码和反码多表示一个最小负数；

(4) 与移码相比，补码的符号位与真值的符号对应关系清楚，符号位为 1 表示负数，符号位为 0 表示正数；

(5) 与反码相比，不需要通过循环移位来调整结果。

现代计算机中带符号整数都用补码表示，故 n 位带符号整数可表示的数值范围为 $-2^{n-1}\sim(2^{n-1}-1)$。例如，8 位带符号整数的表示范围为 $-128\sim+127$。

*2.2.3 C 语言中的整数类型

C 语言中支持多种整数类型。无符号整数在 C 语言中对应 unsigned short、unsigned int(unsigned)、unsigned long 等类型，常在数的后面加一个“u”或“U”来表示，例如，12345U，0x2B3Cu 等；带符号整数在 C 语言中对应 short、int、long 等类型。C 语言中允许无符号整数和带符号整数之间的转换，转换后数的真值是将原二进制机器数按转换后的数据类型重新解释得到。例如，考虑以下 C 代码：

```
1 int x=-1;
2 unsigned u=2147483648;
3
4 printf (“x=%u=%d\n”, x, x);
5 printf (“u=%u=%d\n”, u, u);
```

上述 C 代码中，x 为带符号整数，u 为无符号整数，初值为 2 147 483 648(即 2^{31})。函数 printf 用来输出数值，指示符%u、%d 分别用来表示以无符号整数和带符号整数的形式输出十进制数的值。当在一个 32 位机器上运行上述代码时，它的输出结果如下。

$$x=4\ 294\ 967\ 295=-1$$

$$u=2\ 147\ 483\ 648=-2\ 147\ 483\ 648$$

x 的输出结果说明如下：因为 -1 的补码整数表示为“11…1”，所以当作为 32 位无符号数来解释(格式符为%u)时，其值为 $2^{32}-1=4\ 294\ 967\ 296-1=4\ 294\ 967\ 295$。

u 的输出结果说明如下：2^{31} 的无符号数表示为“100…0”，当被解释为 32 位带符号整数（格式符为%d）时，其值为最小负数：$-2^{32-1}=-2^{31}=-2\ 147\ 483\ 648$（参见前面例 2.12）。

在 C 语言中，如果执行一个运算时同时有无符号数和带符号整数参加，那么，C 编译器会隐含地将带符号整数强制类型转换为无符号数，因而会带来一些意想不到的结果。

表 2.2 给出了一些关系表达式及其在 32 位补码表示机器上的运算结果，1 表示结果为真，0 表示结果为假，其中标有 * 的结果与直觉不符，表中最后一列给出了相应的说明。可以看出，对于同样的 0/1 序列，用不同类型进行解释时的值不同。因为直觉上的取值与计算机中实际取值有差异，因而，有些实际结果与直觉不相符，请注意带 * 的结果说明。

表 2.2　32 位补码表示机器上整数转换示例

关系表达式	运算类型	结果	说　　明
$-2\ 147\ 483\ 648 == 2\ 147\ 483\ 648$U	无符号数	1*	10…0B (2^{31})＝10…0B (2^{31})
$-2\ 147\ 483\ 648 < -2\ 147\ 483\ 647$	带符号数	1	10…0B (-2^{31})＜10…01B ($-2^{31}+1$)
$-2\ 147\ 483\ 648 < 2\ 147\ 483\ 647$U	无符号数	0*	10…0B (2^{31})＞01…1B ($2^{31}-1$)
$-2\ 147\ 483\ 648 < 2\ 147\ 483\ 647$	带符号数	1	10…0B (-2^{31})＜01…1B ($2^{31}-1$)
(unsigned)$-2\ 147\ 483\ 648 < -2\ 147\ 483\ 647$	无符号数	1	10…0B (2^{31})＜10…01B ($2^{31}+1$)
(unsigned)$-2\ 147\ 483\ 648 < 2\ 147\ 483\ 647$	无符号数	0*	10…0B (2^{31})＞01…1B ($2^{31}-1$)

表中第一行的关系表达式中＝＝左边的数据－2 147 483 648 是带符号整数，用补码表示为 10…0。在进行相等比较时，由于“＝＝”右边是一个无符号数，因此，左边的机器数 10…0 也按无符号数解释，其值为 $2\ 147\ 483\ 648=2^{31}$，因而比较结果为 1。这样，看似两个不等的数在计算机内部结果是相等的。在表 2.2 中第三行和第六行也都出现了类似的情况，由此可见，当带符号整数被解释为无符号数时，可能会发生意想不到的结果。

2.3　实数的表示

计算机内部进行数据存储、运算和传送的部件位数有限，因而用定点数表示数值数据时，其表示范围很小。对于 n 位带符号整数，其表示范围为 $-2^{n-1}\sim(2^{n-1}-1)$，运算结果很容易溢出，此外，用定点数也无法表示大量带有小数点的实数。因此，计算机中专门用浮点数来表示实数。

2.3.1　浮点数的表示格式

对于任意一个实数 X，可以表示为

$$X=(-1)^S\times M\times R^E$$

其中 S 取值为 0 或 1，用来决定数 X 的符号，一般用 0 表示正，1 表示负；M 是一个二进制定点小数，称为数 X 的尾数；E 是一个二进制定点整数，称为数 X 的阶码或指数；R 是基数，可以约定为 2、4、16 等。要确定一个实数的值，只要在默认基数 R 下，确定数符 S、尾数 M 和阶码 E 就可以了。因此，浮点数格式只需规定 S、M 和 E 各自所用的位数、编码方式和所在

的位置，而基数 R 与定点数的小数点位置一样，是默认的，不需要明显地表示出来。一般尾数 M 用定点原码小数表示，阶码 E 用移码表示。

在 IEEE 754 浮点数标准被广泛使用之前，不同的计算机所用的浮点数表示格式各不相同。例如，IBM 370 的 32 位短浮点数格式如下。

0	1　　　　7	8　　　　31
数符	阶码	尾数

其中，第 0 位为数符 S；第 1～7 位为 7 位移码表示的阶码 E（偏置常数＝64）；第 8～31 位为 6 位十六进制原码小数表示的尾数 M。基 R 为 16，所以阶码变化 1 等于尾数移动 4 位。

例 2.22　将十进制数 65798 转换为 IBM 370 的 32 位短浮点数格式。

解：因为 $(65798)_{10}=(10106)_{16}=(0.101060)_{16}\times 16^5$。

所以，数符 $S=0$，阶码 $E=(64+5)_{10}=(69)_{10}=(100\ 0101)_2$。

故用该浮点数形式表示如下。

0	100 0101	0001 0000 0001 0000 0110 0000

用十六进制表示为：45 10 10 60H。

例 2.23　将十进制数 65798 转换为下述 32 位浮点数格式。

0	1　　　　8	9　　　　31
数符	阶码	尾 数

其中，第 0 位为数符 S；第 1～8 位为 8 位移码表示的阶码 E（偏置常数为 128）；第 9～31 位为 24 位二进制原码小数表示的尾数。基数为 2，规格化尾数形式为 $\pm 0.1bb\cdots b$，其中第一位“1”不明显表示出来，这样可用 23 个数位表示 24 位尾数。

解：因为 $(65798)_{10}=(1\ 0000\ 0001\ 0000\ 0110)_2=(0.1000\ 0000\ 1000\ 0011\ 0)_2\times 2^{17}$

所以 数符 $S=0$，阶码 $E=(128+17)_{10}=(145)_{10}=(1001\ 0001)_2$

故用该浮点数形式表示如下。

0	100 1000 1	000 0000 1000 0011 0000 0000

用十六进制表示为：48 80 83 00H。

上述格式的规格化浮点数的表示范围如下。

正数最大值：$0.11\cdots 1\times 2^{11\cdots 1}=(1-2^{-24})\times 2^{127}$。

正数最小值：$0.10\cdots 0\times 2^{00\cdots 0}=(1/2)\times 2^{-128}=2^{-129}$。

因为原码是对称的，故该浮点格式的范围是关于原点对称的，如图 2.2 所示。

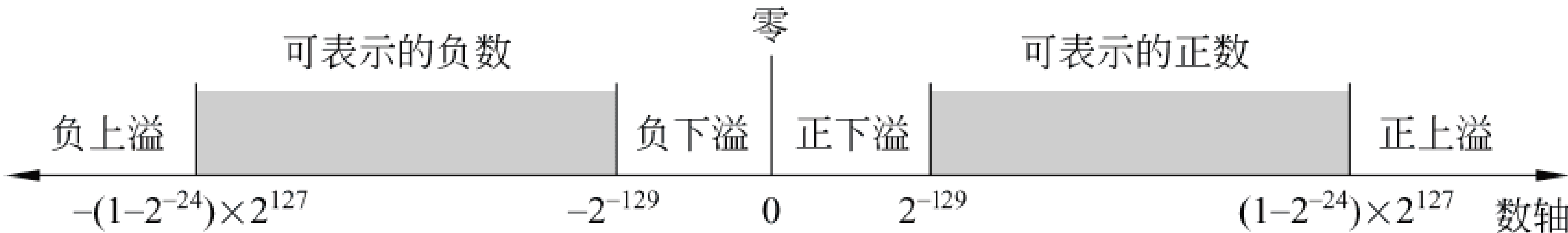


图 2.2　浮点数的表示范围

在图 2.2 中，数轴上有 4 个区间的数不能用浮点数表示。这些区间称为溢出区，接近 0 的区间为下溢区，向无穷大方向延伸的区间为上溢区。

(1) 比 $-(1-2^{-24})\times 2^{127}$ 还小的负数区间是负上溢区。

(2) 比 -2^{-129} 还大的负数区间是负下溢区。

(3) 比 2^{-129} 还小的正数区间是正下溢区。

(4) 比 $(1-2^{-24})\times 2^{127}$ 还大的正数区间是正上溢区。

浮点数表示时有以下几个方面的问题要注意。

1. 0 的表示问题

根据浮点数的表示格式，只要尾数为 0，阶码取任何值其值都为 0，这样的数被称为机器零，因此机器零不唯一。为了唯一地表示机器零，有的计算机规定用一个特定的 0/1 序列来表示机器零。一般用阶码 E 和尾数 M 同时为 0 来表示。即当结果出现尾数为 0 时，不管阶码为何值，都将阶码取为 0。也有的计算机将下溢区（阶码过小）的数近似成机器零。机器零有 +0 和 −0 之分。

2. 浮点数的密度问题

对于 n 位二进制编码，所能表示的不同数据最多有 2^n 个，因此，浮点数虽然表示范围扩大了，但与定点数相比，并没能表示更多的数。实际上只是这些数在数轴上朝正负两个方向在更大的范围内散开，在数轴上的分布变稀疏了。定点数分布是等距且紧密的，而浮点数分布是不等距且稀疏的，越远离原点越稀疏。

3. 表示精度和表示范围的权衡问题

在浮点数总位数不变的情况下，其阶码的位数越多，则尾数位数（有效位数）越少。也即在总位数不变的情况下，若使表示范围扩大，则精度就会变差（数变稀疏）。既增加范围又增加精度的唯一办法就是使用更多的位。所以，大多数计算机都至少提供单精度和双精度两种浮点数格式。浮点数的表数范围和精度除了与阶码的位数和尾数的位数有关外，基数的大小对范围和精度也有影响。基数越大，则范围越大，但精度越低（数变得更稀疏）。因此，对一种固定格式的浮点数而言，更大的基数能给出更大的表示范围，但是，是以牺牲精度为代价的。

2.3.2 浮点数的规格化

浮点数尾数的位数决定浮点数的有效数位，有效数位越多，数据的精度越高。为了在浮点数运算过程中，尽可能多地保留有效数字的位数，使有效数字尽量占满尾数数位，必须在运算过程中对浮点数进行“规格化”操作。对浮点数的尾数进行规格化，除了能得到尽量多的有效数位以外，还可以使浮点数的表示具有唯一性。

从理论上来讲，规格化数的标志是真值的尾数部分中最高位具有非 0 数字。也就是说，若基数为 R，则规格化数的标志是，尾数部分真值的绝对值大于等于 $1/R$。

若浮点数的基数为 2，则尾数规格化的浮点数形式应为 $\pm 0.1bb\cdots b\times 2^E$（这里 b 是 0 或 1）。当尾数用原码表示时，则规格化浮点数的标志为：尾数的数值部分最高位为 1；当尾数用补码表示时，根据规格化数的真值必须具备 $\pm 0.1bb\cdots b$ 的要求可知，当该规格化数为正数时，尾数的补码必为 $0.1bb\cdots b$，而当规格化数为负数（即 $-0.1bb\cdots b$）时，则除了 -0.1

(即−1/2)以外,其补码必为 1.0bb…b 的形式。因此,补码表示的规格化数的标志为:尾数的符号位和最高数值位相异。这种规定简化了补码表示规格化数的判断过程,但使得−1/2被排除在规格化数的范围之外。−1/2 的补码形式为 1.10…0,符号位和最高数值位相同,所以按补码尾数规格化形式的规定,它不是规格化数。但是,因为补码可以表示最小负数−1,所以遇到这种情况时,可将尾数−1/2 乘以 2,使尾数变为−1,阶码减 1,即左规一次。此时,因为−1 的补码为 1.00…0,所以满足规格化形式的要求。

规格化操作有两种:“左规”和“右规”。当有效数位进到小数点前面时,需要进行右规。右规时,尾数每右移一位,阶码加 1,直到尾数变成规格化形式为止,右规时阶码会增加,因此阶码有可能溢出;当出现形如 $\pm 0.0\cdots 0bb\cdots b\times 2^E$ 的运算结果时,需要进行左规,左规时,尾数每左移一位,阶码减 1,直到尾数变成规格化形式为止。

2.3.3 IEEE 754 浮点数标准

直到 20 世纪 80 年代初,浮点数表示格式还没有统一标准,不同厂商计算机内部浮点数表示格式不同,在不同结构的计算机之间进行数据传送或程序移植时,必须进行数据格式的转换,而且还由于数据格式转换而带来运算结果的不一致。因而,20 世纪 70 年代后期,IEEE 成立委员会着手制定浮点数标准,1985 年完成了浮点数标准 IEEE 754 的制定。

IEEE 754 标准的主要起草者是加州大学伯克利分校数学系教授 William Kahan,他帮助 Intel 公司设计了 8087 浮点处理器(FPU),并以此为基础形成了 IEEE 754 标准,Kahan 教授也因此获得了 1987 年的图灵奖。

目前几乎所有计算机都采用 IEEE 754 标准表示浮点数。在这个标准中,提供了两种基本浮点格式:32 位单精度和 64 位双精度格式,如图 2.3 所示。

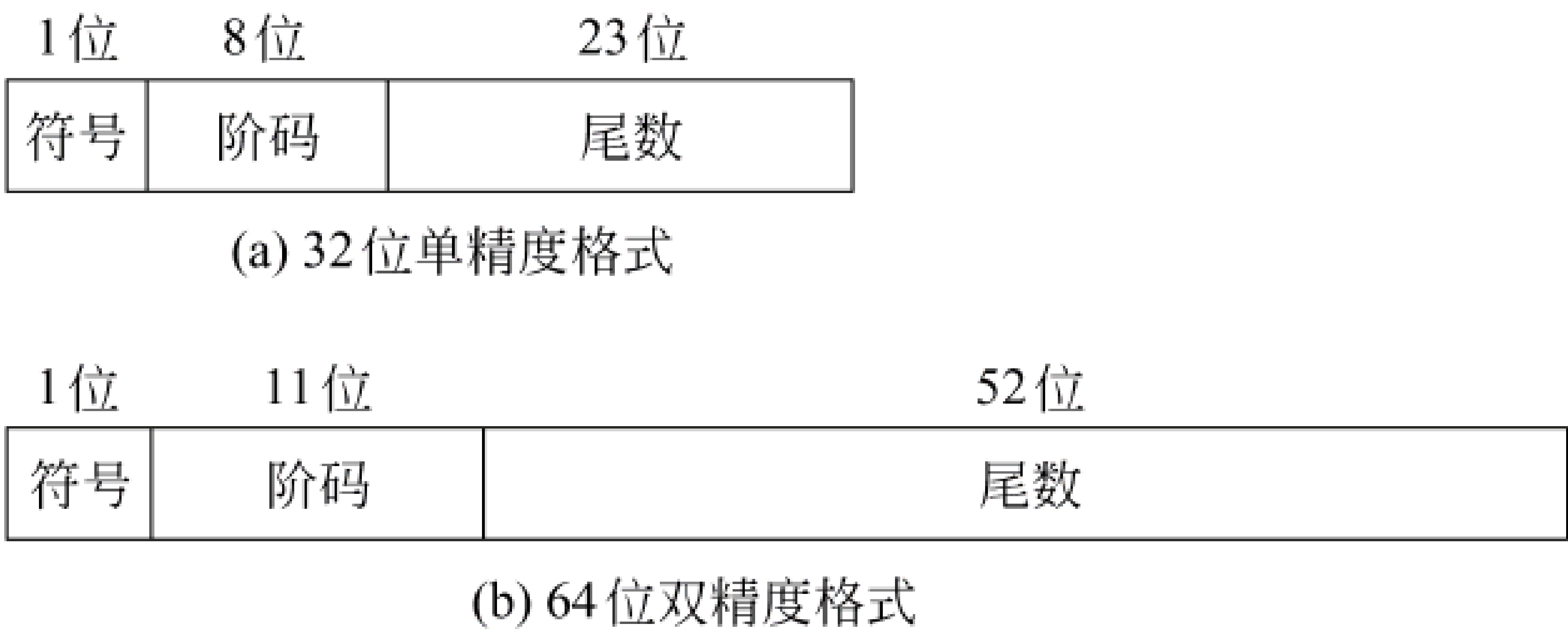


图 2.3 IEEE 754 浮点数格式

32 位单精度格式中包含一位符号 s、8 位阶码 e 和 23 位尾数 f;64 位双精度格式包含一位符号 s、11 位阶码 e 和 52 位尾数 f。其基数隐含为 2;尾数用原码表示,第一位总为 1,因而可在尾数中省略第一位的 1,称为隐藏位,使得单精度格式的 23 位尾数实际上表示了 24 位有效数字,双精度格式的 52 位尾数实际上表示了 53 位有效数字。IEEE 754 规定隐藏位 1 的位置在小数点之前。

在 IEEE 754 标准中,阶码用移码表示,但偏置常数并不是通常 n 位移码所用的 2^{n-1},而是$(2^{n-1}-1)$,因此,单精度和双精度浮点数的偏置常数分别为 127 和 1023。因为尾数中有一位在小数点之前的 1 在隐藏位中,所以,如果尾数换成用等值的纯小数表示的话,阶码就需要加 1,相当于偏置常数为 128 和 1024。因此,最终计算结果与采用通常的偏置常数的情

况是相同的。IEEE 754 的这种“尾数带一个隐藏位，偏置常数用($2^{n-1}-1$)”的做法，不仅没有改变传统做法的计算结果，而且带来了以下两个好处：

(1) 尾数可表示的位数多一位，因而使浮点数的精度更高。

(2) 阶码的可表示范围更大，因而使浮点数范围更大。例如，对于单精度浮点数格式，其阶码为 8 位，当偏置常数用 128 时，最大机器码 1111 1111 对应的值为 255－128＝127；当偏移常数用 127 时，其对应的值为 255－127＝128。显然，偏置常数采用 127 时阶码的范围更大。

对于 IEEE 754 标准格式的数，一些特殊的位序列(如阶码为全 0 或全 1)有其特别的解释。表 2.3 给出了对各种形式的数的解释。

表 2.3　IEEE 754 浮点数的解释

值的类型	单精度(32 位)				双精度(64 位)			
	符号	阶　码	尾数	值	符号	阶　码	尾数	值
正零	0	0	0	0	0	0	0	0
负零	1	0	0	－0	1	0	0	－0
正无穷大	0	255(全 1)	0	∞	0	2047(全 1)	0	∞
负无穷大	1	255(全 1)	0	$-\infty$	1	2047(全 1)	0	$-\infty$
无定义数	0 或 1	255(全 1)	$\neq 0$	NaN	0 或 1	2047(全 1)	$\neq 0$	NaN
规格化非零正数	0	$0<e<255$	f	$2^{e-127}(1.f)$	0	$0<e<2047$	f	$2^{e-1023}(1.f)$
规格化非零负数	1	$0<e<255$	f	$-2^{e-127}(1.f)$	1	$0<e<2047$	f	$-2^{e-1023}(1.f)$
非规格化正数	0	0	$f\neq 0$	$2^{-126}(0.f)$	0	0	$f\neq 0$	$2^{-1022}(0.f)$
非规格化负数	1	0	$f\neq 0$	$-2^{-126}(0.f)$	1	0	$f\neq 0$	$-2^{-1022}(0.f)$

在表 2.3 中，对 IEEE 754 中规定的数进行了以下分类。

1. 全 0 阶码全 0 尾数：＋0/－0

IEEE 754 的 0 有两种表示：＋0 和－0。0 的符号取决于数的符号位 s。一般情况下＋0 和－0 是等效的。有些计算机将运算结果在下溢区(阶码过小)的数近似成 0。正下溢区的数为＋0，负下溢区的数为－0。

2. 全 0 阶码非 0 尾数：非规格化数

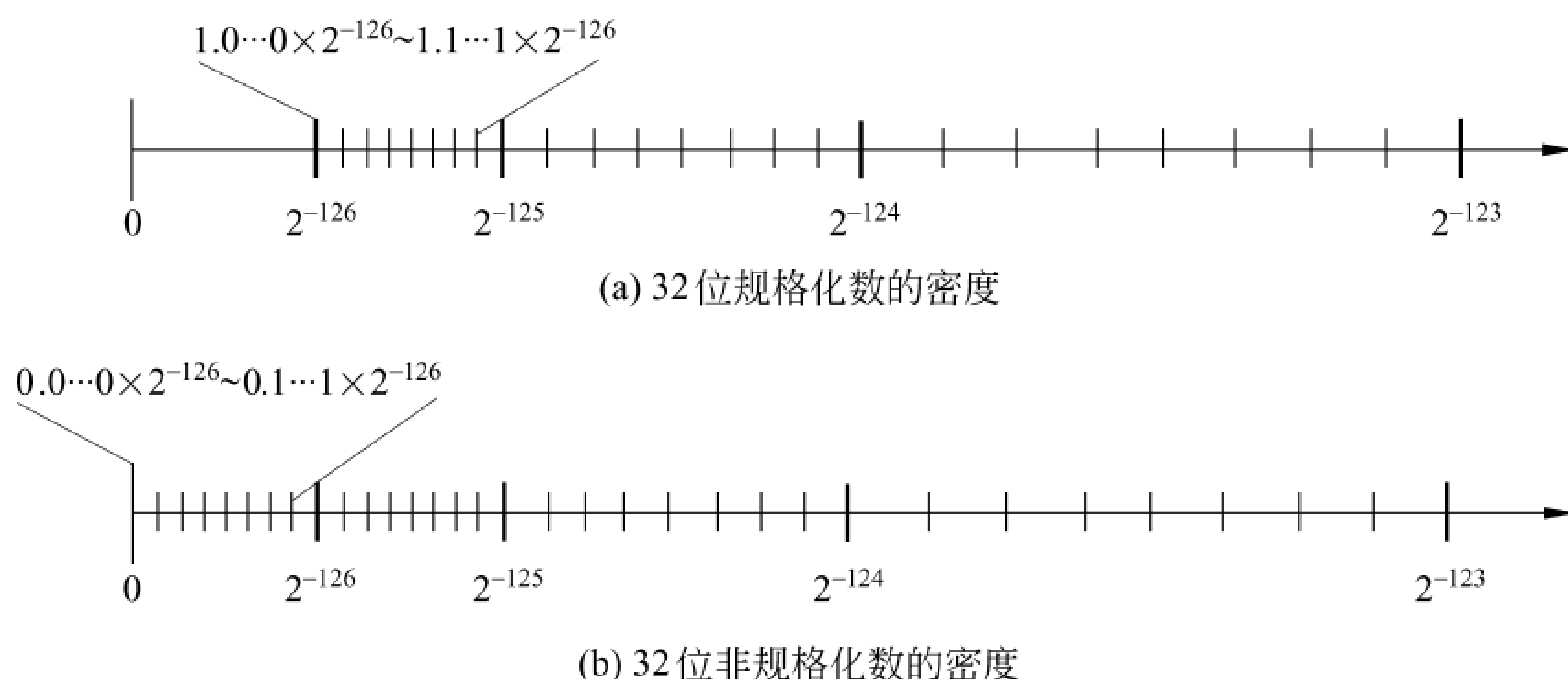
非规格化数的特点是阶码部分的编码为全 0，尾数高位有一个或几个连续的 0，但不全为 0。因此非规格化数的隐藏位为 0，并且单精度和双精度浮点数的阶码的值分别为－126 或－1022，故数值分别为$(-1)^s\times 0.f\times 2^{-126}$和$(-1)^s\times 0.f\times 2^{-1022}$。

非规格化数可用于处理阶码下溢，使得出现比最小规格化数还小的数时程序也能继续进行下去。当运算结果的阶码太小(比最小能表示的阶码还小，即小于－126 或小于－1022)时，尾数右移一次，阶码加 1，如此循环直到尾数为 0 或阶码达到可表示的最小值(－126 或－1022)。这个过程称为“逐级下溢”。因此，“逐级下溢”的结果就是使尾数变为非规格化数，阶码变为最小负数。例如，当一个十进制运算系统的最小阶码为－99 时，以下情况需进行阶码逐级下溢。

$$2.0000\times10^{-26}\times5.2000\times10^{-84}=1.04\times10^{-109}\rightarrow0.104\times10^{-108}\rightarrow0.0104\times10^{-107}\rightarrow0.0010\times10^{-106}\rightarrow0.0001\times10^{-105}\rightarrow0.0000\times10^{-104}\rightarrow0.0$$

$$2.0002\times10^{-98}-2.0000\times10^{-98}=2.0000\times10^{-102}\rightarrow0.2000\times10^{-101}\rightarrow0.0200\times10^{-100}\rightarrow0.0020\times10^{-99}$$

图 2.4 表示加入非规格化数后 IEEE 754 的表数范围的变化。图中将可表示数以 $[2^n,2^{n+1}]$ 的区间分组。区间 $[2^n,2^{n+1}]$ 内所有数的阶码相同，都为 n，而尾数部分的变化范围为 1.00…0～1.11…1，这里小数点前的 1 是隐藏的隐含位。对于 32 位单精度规格化数，因为尾数的位数有 23 位，故每个区间内的数的个数相同，都是 2^{23} 个。例如，在正数范围内最左边的区间为 $[2^{-126},2^{-125}]$，在该区间内，最小规格化数为 $1.00\cdots0\times2^{-126}$，最大规格化数为 $1.11\cdots1\times2^{-126}$。在该区间中的各个相邻数之间具有等距性，其距离为 $2^{-23}\times2^{-126}$，该区间右边相邻的区间为 $[2^{-125},2^{-124}]$，区间内各相邻数间的距离为 $2^{-23}\times2^{-125}$。由此可见，每个右区间内相邻数间的距离总比左边一个区间的相邻数距离大一倍，因此越离原点近的区间内的数的间隙越小。图 2.4(a)给出了未定义非规格化数的 32 位单精度浮点数的情况。在图中可看出，在 0 和最小规格化数 2^{-126} 之间有一个间隙未被利用。定义了非规格化数后，在 0 和 2^{-126} 之间就增加了 2^{23} 个附加数，这些相邻附加数之间与区间 $[2^{-126},2^{-125}]$ 内的相邻数等距。附加的 2^{23} 个数为非规格化数，所有这些数据具有与区间 $[2^{-126},2^{-125}]$ 内的数相同的阶码，即最小阶码(−126)。尾数部分的变化范围为 0.00…0～0.11…1。这里的隐含位为 0，这也是非规格化数的重要标志之一。



(a) 32 位规格化数的密度

(b) 32 位非规格化数的密度

图 2.4　IEEE 754 中加入非规格化数后表示范围的变化

3. 全 1 阶码全 0 尾数：+∞/−∞

引入无穷大数使得在计算过程出现异常的情况下程序能继续进行下去，并且可为程序提供错误检测功能。+∞在数值上大于所有有限数，−∞则小于所有有限数，无穷大数既可作为操作数，也可能是运算的结果。当操作数为无穷大时，系统可以有两种处理方式。

(1) 产生不发信号的非数 NaN。

如+∞+(−∞)，+∞−(+∞)，∞/∞等。

(2) 产生明确的结果。

如 5+(+∞)=+∞，(+∞)+(+∞)=+∞，5−(+∞)=−∞，(−∞)−(+∞)=−∞等。

4. 全 1 阶码非 0 尾数：NaN（Not a Number）

NaN（Not a Number）表示一个没有定义的数，称为非数。分为不发信号（quiet）和发信号（signaling）两种非数。有的书中把它们分别称为"静止的 NaN"和"通知的 NaN"。

表 2.4 给出了能产生不发信号（静止的）NaN 的计算操作。

表 2.4 产生不发信号 NaN 的操作

运算类型	产生不发信号 NaN 的计算操作
所有	对通知 NaN 的任何计算操作
加减	无穷大相减： $(+\infty)+(-\infty)$ $(+\infty)-(+\infty)$ $(-\infty)+(+\infty)$ $(-\infty)-(-\infty)$
乘	$0\times\infty$
除	$0/0$ 或 ∞/∞
求余	x MOD 0 或 ∞ MOD y
平方根	$\sqrt{x}$ 且 $x<0$

引入 NaN 的目的是为了检测非初始化值的使用。程序员或编译程序可用非数表示每个变量的非初始化值。引入 NaN 还可以使计算出现异常时程序能继续进行下去，让程序员将测试或判断延迟到方便的时候进行。可用尾数取值的不同来区分是"不发信号 NaN"还是"发信号 NaN"。当最高有效位为 1 时，为不发信号（静止的）NaN，当结果产生这种非数时，不发"异常"通知，即不进行异常处理；当最高有效位为 0 时为发信号（通知的）NaN，当结果产生这种非数时，则发一个异常操作通知，表示要进行异常处理。因为 NaN 的尾数是非 0 数，除了第一位有定义外其余的位没有定义，所以可用其余位来指定具体的异常条件。一些没有数学解释的计算（如 $0/0$，$0\times\infty$ 等）会产生一个非数 NaN。

5. 阶码非全 0 且非全 1：规格化非 0 数

对于阶码范围在 1～254（单精度）和 1～2046（双精度）的数，是一个正常的规格化非 0 数。根据 IEEE 754 的定义，这种数的阶码的真值范围应该是 $-126\sim+127$（单精度）和 $-1022\sim+1023$（双精度），其值的计算公式分别为：

$$((-1)^s\times 1.f\times 2^{e-127}) \quad 和 \quad ((-1)^s\times 1.f\times 2^{e-1023})$$

例 2.24 将十进制数 -0.75 转换为 IEEE 754 的单精度浮点数格式表示。

解：$(-0.75)_{10}=(-0.11)_2=(-1.1)_2\times 2^{-1}=(-1)^s\times 1.f\times 2^{e-127}$。

所以 $s=1$，$f=0.100\cdots0$，$e=(127-1)_{10}=(126)_{10}=(0111\ 1110)_2$。

规格化浮点数表示为 1 0111 1110 1000 0000…0000 000。

用十六进制表示为 BF 40 00 00H。

例 2.25 求 IEEE 754 单精度浮点数 C0 A0 00 00H 的值是多少。

解：求一个机器数的真值，就是将该数转换为十进制数。

首先将 C0 A0 00 00H 展开为一个 32 位单精度浮点数：1 10000001 010 0000…0000。

根据 IEEE 754 单精度浮点数格式，知：

符号 $s=1$，尾数 $f=(0.01)_2=(0.25)_{10}$，阶码 $e=(10000001)_2=(129)_{10}$。

所以，其值为 $(-1)^s\times 1.f\times 2^{e-127}=(-1)^1\times 1.25\times 2^{129-127}=-1.25\times 2^2=-5.0$。

IEEE 754 标准的单精度和双精度格式的特征参数见表 2.5。

表 2.5 IEEE 754 浮点数格式参数

参　　数	单精度浮点数	双精度浮点数
字宽(位数)	32	64
阶码宽度(位数)	8	11
阶码偏置常数	127	1023
最大阶码	127	1023
最小阶码	−126	−1022
尾数宽度	23	52
阶码个数	254	2046
尾数个数	2^{23}	2^{52}
值的个数	1.98×2^{31}	1.99×2^{63}
数的量级范围	$10^{-38}\sim 10^{+38}$	$10^{-308}\sim 10^{+308}$

IEEE 754 用全 0 阶码和全 1 阶码表示一些特殊值，如 0、∞和 NaN，因此，除去全 0 和全 1 阶码后，单精度和双精度格式的阶码个数分别为 254 和 2046，最大阶码的值也相应地变为 127 和 1023。单精度规格化数的个数为 $254\times 2^{23}=1.98\times 2^{31}$，双精度规格化数的个数为 $2046\times 2^{52}=1.99\times 2^{63}$。根据单精度和双精度格式的最大阶码分别为 127 和 1023，可以得出数的量级范围分别为 $10^{-38}\sim 10^{+38}$ 和 $10^{-308}\sim 10^{+308}$。

IEEE 754 除了对上述单精度和双精度浮点数格式进行了具体的规定以外，还对单精度扩展和双精度扩展两种格式的最小长度和最小精度进行了规定。例如，IEEE 754 规定，双精度扩展格式必须至少具有 64 位有效数字，并总共占用至少 79 位，但没有规定其具体的格式，处理器厂商可以选择符合该规定的格式。

例如，Intel x86 FPU 采用 80 位双精度扩展格式，包含 4 个字段：一位符号位 s、15 位阶码 e(偏置常数为 16383)、一位显式首位有效位(explicit leading significand bit) j 和 63 位尾数 f。Intel 采用的这种扩展浮点数格式与 IEEE 754 规定的单精度和双精度浮点数格式的一个重要的区别是，它没有隐藏位，有效位数共 64 位。Intel 安腾 FPU 采用 82 位扩展精度。

又如，SPARC 和 Power PC 处理器中采用 128 位扩展双精度浮点数格式，包含一位符号位 s、15 位阶码 e(偏置常数为 16383)和 112 位尾数 f，采用隐藏位，所以有效位数为 113 位。

*2.3.4 C 语言中的浮点数类型

C 语言中有 float 和 double 两种不同浮点数类型，分别对应 IEEE 754 单精度浮点数格式和双精度浮点数格式，相应的十进制有效数字分别为 7 位和 17 位。

C 对于扩展双精度的相应类型是 long double，但是 long double 的长度和格式随编译器和处理器类型的不同而有所不同。例如，Microsoft Visual C++ 6.0 版本以下的编译器都不支持该类型，因此，用其编译出来的目标代码中 long double 和 double 一样，都是 64 位双精度；在 IA-32 上使用 gcc 编译器时，long double 类型数据采用 2.3.3 节中所述的 Intel x86 FPU 的 80 位双精度扩展格式表示；在 SPARC 和 Power PC 处理器上使用 gcc 编译器时，long double 类型数据采用 2.3.3 节中所述的 128 位双精度扩展格式表示。

当在 int、float 和 double 等类型数据之间进行强制类型转换时，程序将得到以下数值转换结果（假定 int 为 32 位）。

（1）从 int 转换为 float 时，不会发生溢出，但可能有数据被舍入。

（2）从 int 或 float 转换为 double 时，因为 double 的有效位数更多，故能保留精确值。

（3）从 double 转换为 float 时，因为 float 表示范围更小，故可能发生溢出，此外，由于有效位数变少，故可能被舍入。

（4）从 float 或 double 转换为 int 时，因为 int 没有小数部分，所以数据可能会向 0 方向被截断。例如，1.9999 被转换为 1，－1.9999 被转换为－1。此外，因为 int 的表示范围更小，故可能发生溢出。将大的浮点数转换为整数可能会导致程序错误，这在历史上曾经有过惨痛的教训。

1996 年 6 月 4 日，Ariana 5 火箭初次航行，在发射仅仅 37 秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值 5 亿美元的通信卫星。根据调查发现，原因是控制惯性导航系统的计算机向控制引擎喷嘴的计算机发送了一个无效数据。它没有发送飞行控制信息，而是发送了一个异常诊断位模式数据，表明在将一个 64 位浮点数转换为 16 位带符号整数时，产生了溢出异常。溢出的值是火箭的水平速率，这比原来的 Ariana 4 火箭所能达到的速率高出了 5 倍。在设计 Ariana 4 火箭软件时，设计者确认水平速率决不会超出一个 16 位的整数，但在设计 Ariana 5 时，他们没有重新检查这部分，而是直接使用了原来的设计。

在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误，这种错误有时会带来重大损失，因此，编程时要非常小心。

例 2.26 假定变量 i、f、d 的类型分别是 int、float 和 double，它们可以取除 $+\infty$、$-\infty$ 和 NaN 以外的任意值。请判断下列每个 C 语言关系表达式在 32 位机器上运行时是否永真。

```
A. i==(int)(float) i
B. f==(float)(int)f
C. i==(int)(double)i
D. f==(float)(double)f
E. d==(float)d
F. f==-(-f)
G. (d+f)-d==f
```

解：A. 不是，int 精度比 float 高，当 i 转换为 float 后再到 int 时，有效位数可能丢失。

B. 不是，float 有小数部分，当 f 转换为 int 后再到 float 时，小数部分可能会丢失。

C. 是，double 比 int 有更大的精度和范围，当 i 转换为 double 后再到 int 时数值不变。

D. 是，double 比 float 有更大的精度和范围，当 f 转为 double 后再到 float 时数值不变。

E. 不是，double 比 float 有更大的精度和范围，当 d 转换为 float 后数值可能改变。

F. 是，浮点数取负就是简单将数符取反。

G. 不是，例如，当 $d=1.79\times10^{308}$、$f=1.0$ 时，左边为 0(因为 $d+f$ 时 f 需向 d 对阶，对阶后 f 的尾数有效数位被舍去而变为 0，故 $d+f$ 仍然等于 d，再减去 d 后结果为 0)，而右边为 1。

2.4 十进制数的表示

人们日常使用和熟悉的是十进制，使用计算机来处理数据时，在计算机外部(如键盘输入、屏幕显示或打印输出)看到的数据基本上是十进制形式，因此，有时需要计算机内部能够表示和处理十进制数据，以方便直接进行十进制数的输入输出或直接用十进制数进行计算。

在计算机内部，可以采用数字 0～9 对应的 ASCII 码字符来表示十进制数，也可以采用二进制编码的十进制数(Binary Coded Decimal)来表示十进制数。

*2.4.1 用 ASCII 码字符表示

为方便十进制数输入输出(如打印或显示)，可以把十进制数看成字符串，直接用 ASCII 码表示，0～9 分别对应 30H～39H。这种表示方式下，1 位十进制数对应 8 位二进制数。

一个十进制数在计算机内部需占用多个连续字节，因此，在存取一个十进制数时，必须说明该十进制数在内存的起始地址和字节个数。根据不同的数符表示方式，可以分为前分隔数字串和后嵌入数字串两种格式。

***1. 前分隔数字串**

前分隔数字串方式是：将符号位单独用一个字节来表示，位于数字串之前。正号用字符“+”的 ASCII 码(2BH)表示；负号用字符“−”的 ASCII 码(2DH)表示。例如：十进制数 +236 表示为 0010 1011 0011 0010 0011 0011 0011 0110B(2B 32 33 36H)，在内存中占用 4 个字节；十进制数 −2369 表示为 0010 1101 0011 0010 0011 0011 0011 0110 0011 1001B(2D 32 33 36 39H)，在内存中占用 5 个字节。

***2. 后嵌入数字串**

后嵌入数字串方式为：符号位不单独用一个字节来表示，而是嵌入到最低一位数字的 ASCII 码中。正数的最低一位数字编码不变；负数的最低一位数字编码的高 4 位由原来的 0011 变为 0111。例如，十进制数 +236 表示为 0011 0010 0011 0011 0011 0110B(32 33 36H)，在内存中占用 3 个字节；十进制数 −2369 表示为 0011 0010 0011 0011 0011 0110 0111 1001B(32 33 36 79H)，在内存中占用 4 个字节。

可以明显看出，后嵌入数字串方式比前分隔数字串方式少占用一个字节，节省了存储空间。但是，对于负数来说，最低一位数字不是采用十进制数字的 ASCII 码，所以，在显示或打印前必须先转换成可打印字符编码。

用 ASCII 码字符串方式来表示十进制数，方便了十进制数的输入输出，但是，由于这种表示形式中含有非数值信息(高 4 位编码)，所以对十进制数的运算很不方便。如果要对这

种形式的十进制数进行计算，则必须先转换为二进制数或用 BCD 码表示十进制数。

2.4.2 用 BCD 码表示

计算机内部，通常情况下都用二进制数进行表示和运算，只有非常特殊的情况才用十进制数进行表示和运算。例如，在数据输入输出量很大的商用领域，为了减少大量的十进制数和二进制数之间的转换，可采用直接对十进制数进行运算的方式。这种十进制数用二进制编码的形式，通过专门的十进制数运算指令进行处理。计算机中可有专门的逻辑线路在 BCD 码运算时使每 4 位二进制数按十进制进行处理。

每位十进制数的取值可以是 0～9 这 10 个数之一，因此，每一个十进制数位必须至少有 4 位二进制位来表示。而 4 位二进制位可以组合成 16 种状态，去掉 10 种状态后还有 6 种冗余状态，所以从 16 种状态中选取 10 种状态表示十进制数位 0～9 的方法很多，可以产生多种 BCD 码。

*** 1. 有权 BCD 码**

十进制有权码是指表示每个十进制数位的 4 个二进制数位(称为基 2 码)都有一个确定的权。表 2.6 列出了几种常用的十进制有权码方案。

表 2.6 4 位有权码

十进制数	8421 码	2421 码	5211 码	84-2-1 码	4311 码
0	0000	0000	0000	0000	0000
1	0001	0001	0001	0111	0001
2	0010	0010	0011	0110	0011
3	0011	0011	0101	0101	0100
4	0100	0100	0111	0100	1000
5	0101	1011	1000	1011	0111
6	0110	1100	1010	1010	1011
7	0111	1101	1100	1001	1100
8	1000	1110	1110	1000	1110
9	1001	1111	1111	1111	1111

在表 2.6 中，最常用的一种编码就是 8421 码，它选取 4 位二进制数按计数顺序的前 10 个代码与十进制数字相对应，每位的权从左到右分别为 8、4、2、1，因此称为 8421 码，也称自然(Nature) BCD 码，记为 NBCD 码。又如，84-2-1 码各位的权从左到右分别为 8、4、－2、－1。

*** 2. 无权 BCD 码**

十进制无权码是指表示每个十进制数位的 4 个基 2 码没有确定的权。在无权码方案中，用得较多的是余 3 码和格雷码。表 2.7 列出了这两种常用的十进制无权码方案。

余 3 码是在 8421 码的基础上，把每个代码都加 0011 而形成的。其主要优点是执行十进制数加法时，能正确产生进位，而且还给减法运算带来方便。

表 2.7　4 位无权码

十进制数	余 3 码	格雷码(1)	格雷码(2)
0	0011	0000	0000
1	0100	0001	0100
2	0101	0011	0110
3	0110	0010	0010
4	0111	0110	1010
5	1000	1110	1011
6	1001	1010	0011
7	1010	1000	0001
8	1011	1100	1001
9	1100	0100	1000

格雷码又称循环码。其编码规则是使任何两个相邻的代码只有一个二进位的状态不同,其余三个二进位必须相同。这样使得从一个编码变到下一个编码时,只有一位发生变化,变码速度最快,且有利于得到更好的译码波形,故在 D/A 或 A/D 转换电路中得到很好的运行结果。

一个十进制数用多个对应的 BCD 码组合表示,每个数字对应 4 位 BCD 码,两个数字占一个字节,数符可用 1 位二进制表示,1 表示负数,0 表示正数;也可用 4 位二进制表示,并放在数字串最后,通常用 1100 表示正号,用 1101 表示负号。例如,Pentium 处理器中的十进制数占 80 位,第一个字节中的最高位为符号位,后面的 9 个字节可表示 18 位十进制数。

使用 BCD 码会耗费较多的设备量。例如,处理 1000 个信息,需要 3 位十进制数,因而至少需要$(3+1)\times4=16$位的设备量,而对于二进制系统,因为$2^{10}=1024$,所以只需 10 位的设备量就足够了。因此,在计算机内部,通常情况下都用二进制数进行数据的表示和运算。

2.5　非数值数据的编码表示

逻辑值、字符等数据都是非数值数据,在机器内部它们也用二进制表示。下面分别介绍这些非数值数据的编码表示。

2.5.1　逻辑值

正常情况下,每个字或其他可寻址单位(字节、半字等)是作为一个整体数据单元看待的。但是,某些时候还需要将一个 n 位数据看成是由 n 个一位数据组成,每个取值为 0 或 1。例如,有时需要存储一个布尔值或二进制数据阵列,阵列中的每项只能取值为 1 或 0;有时可能需要提取一个数据项中的某位进行诸如“置位”或“清 0”等操作。当数据以这种方式看待时,就被认为是逻辑数据。因此 n 位二进制数可表示 n 个逻辑值。逻辑数据只能参加

逻辑运算,并且是按位进行的,如按位“与”、按位“或”、逻辑左移、逻辑右移等。

逻辑数据和数值数据都是一串 0/1 序列,在形式上无任何差异,需要通过指令的操作码类型来识别它们。例如,逻辑运算指令处理的是逻辑数据,算术运算指令处理的是数值数据。

2.5.2 西文字符

西文由拉丁字母、数字、标点符号及一些特殊符号所组成,它们统称为“字符”(character)。所有字符的集合叫做“字符集”。字符不能直接在计算机内部进行处理,因而也必须对其进行数字化编码,字符集中每一个字符都有一个代码(二进制编码的 0/1 序列),构成了该字符集的代码表,简称码表。码表中的代码具有唯一性。

字符主要用于外部设备和计算机之间交换信息。一旦确定了所使用的字符集和编码方法后,计算机内部所表示的二进制代码和外部设备输入、打印和显示的字符之间就有唯一的对应关系。

字符集有多种,每一个字符集的编码方法也多种多样。目前计算机中使用最广泛的西文字符集及其编码是 ASCII 码,即美国标准信息交换码(American Standard Code for Information Interchange),ASCII 字符编码见表 2.8。

表 2.8 ASCII 码表

	$b_6b_5b_4$ =000	$b_6b_5b_4$ =001	$b_6b_5b_4$ =010	$b_6b_5b_4$ =011	$b_6b_5b_4$ =100	$b_6b_5b_4$ =101	$b_6b_5b_4$ =110	$b_6b_5b_4$ =111
$b_3b_2b_1b_0$ =0000	NUL	DLE	SP	0	@	P	`	p
$b_3b_2b_1b_0$ =0001	SOH	DC1	!	1	A	Q	a	q
$b_3b_2b_1b_0$ =0010	STX	DC2	"	2	B	R	b	r
$b_3b_2b_1b_0$ =0011	ETX	DC3	#	3	C	S	c	s
$b_3b_2b_1b_0$ =0100	EOT	DC4	$	4	D	T	d	t
$b_3b_2b_1b_0$ =0101	ENQ	NAK	%	5	E	U	e	u
$b_3b_2b_1b_0$ =0110	ACK	SYN	&	6	F	V	f	v
$b_3b_2b_1b_0$ =0111	BEL	ETB	'	7	G	W	g	w
$b_3b_2b_1b_0$ =1000	BS	CAN	(	8	H	X	h	x
$b_3b_2b_1b_0$ =1001	HT	EM	)	9	I	Y	i	y
$b_3b_2b_1b_0$ =1010	LF	SUB	*	:	J	Z	j	z
$b_3b_2b_1b_0$ =1011	VT	ESC	+	;	K	[	k	{
$b_3b_2b_1b_0$ =1100	FF	FS	,	<	L	\	l	|
$b_3b_2b_1b_0$ =1101	CR	GS	-	=	M	]	m	}
$b_3b_2b_1b_0$ =1110	SO	RS	.	>	N	^	n	~
$b_3b_2b_1b_0$ =1111	SI	US	/	?	O	_	o	Del

从表2.8中可看出每个字符都由7个二进位 $b_6b_5b_4b_3b_2b_1b_0$ 表示，其中 $b_6b_5b_4$ 是高位部分，$b_3b_2b_1b_0$ 是低位部分。一个字符在计算机中实际上是用8位表示的。一般情况下，最高一位 b_7 为0。在需要奇偶校验时，这一位可用于存放奇偶校验值，此时称这一位为奇偶校验位。7个二进位 $b_6b_5b_4b_3b_2b_1b_0$ 从0000000到1111111共表示128种编码，可用来表示128个不同的字符，其中包括10个数字、26个小写字母、26个大写字母、算术运算符、标点符号、商业符号等。表中共有95个可打印(或显示)字符和33个控制字符。这95个可打印(或显示)字符在计算机键盘上能找到相应的键，按键后就可将对应字符的二进制编码送入计算机内。表中第0列和第1列以及第7列最末一个字符(Del)称为控制字符，共33个，它们在传输、打印或显示输出时起控制作用。

从表2.8中可看出ASCII字符编码有两个规律。

(1) 字符0～9这10个数字字符的高三位编码为011，低4位分别为0000～1001。当去掉高3位时，低4位正好是0～9这10个数字的8421码。这样既满足了正常的排序关系，又有利于实现ASCII码与十进制数之间的转换。

(2) 英文字母字符的编码值也满足正常的字母排序关系，而且大、小写字母的编码之间有简单的对应关系，差别仅在 b_5 这一位上，若这一位为0，则是大写字母；若为1，则是小写字母。这使得大、小写字母之间的转换非常方便。

西文字符集的编码不止ASCII码一种，较常用的还有一种是用8位二进制数表示一个字符的EBCDIC码(Extended Binary Coded Decimal Interchange Code)，该码共有256个编码状态，最多可表示256个字符，但这256个编码并没有全部被用来表示字符。0～9这10个数字字符的高4位编码为1111，低4位还是分别对应0000～1001。大、小写英文字母字符的编码也是仅一位(第2位)不同。

*2.5.3 汉字字符

西文是一种拼音文字，用有限的几个字母可以拼写出所有单词。因此西文中仅需要对有限个少量的字母和一些数学符号、标点符号等辅助字符进行编码，所有西文字符集的字符总数不超过256个，所以使用7个或8个二进位就可表示。中文信息的基本组成单位是汉字，汉字也是字符。但汉字是表意文字，一个字就是一个方块图形。计算机要对汉字信息进行处理，就必须对汉字本身进行编码，但汉字的总数超过6万字，数量巨大，给汉字在计算机内部的表示、汉字的传输与交换、汉字的输入和输出等带来了一系列问题。为了适应汉字系统各组成部分对汉字信息处理的不同需要，汉字系统必须处理以下几种汉字代码：输入码、内码、字模点阵码。

1. 汉字的输入码

由于计算机最早是由西方国家研制开发的，最重要的信息输入工具——键盘——是面向西文设计的，一个或两个西文字符对应着一个按键，非常方便。但汉字是大字符集，专门的汉字输入键盘由于键多、查找不便、成本高等原因而几乎无法采用。怎样向计算机输入汉字呢？一种是手写汉字联机识别输入，或者是印刷汉字扫描输入后自动识别，这两种方法现均已达到实用水平。现在还有一种用语音输入汉字的方法，虽然简单易操作，但离实用阶段还相差很远。目前来说，最简便、最广泛采用的汉字输入方法是利用英文键盘输入汉字。由于汉字字数多，无法使每个汉字与西文键盘上的一个键相对应，因此必须使每个汉字用一个或几个键来表示，这种对每个汉字用相应的按键进行的编码表示就称为汉字的“输入码”，又

称外码。因此汉字的输入码的码元(即组成编码的基本元素)是西文键盘中的某个按键。

汉字的输入编码方案有几百种之多。能够被广泛接受的编码方案应具有下列特点：易学习、易记忆、效率高(击键次数较少)、重码少、容量大(包含汉字的字数多)等。到目前为止，还没有一种在所有方面都很好的编码方法，真正推广应用较好的也只有少数几种。汉字输入编码方法大体分成4类。(1)数字编码：用一串数字来表示汉字的编码。例如电报码、区位码等，它们难以记忆，不易推广。(2)字音编码：基于汉语拼音的编码，它简单易学，适合于非专业人员。缺点是同音字引起的重码多，需增加选择操作。例如，现在常用的微软拼音输入法和智能ABC输入法等。(3)字形编码：将汉字的字形分解归类而给出的编码，它重码少、输入速度快，但编码规则不易掌握，例如，五笔字型法和表形码就是这类编码。(4)形音编码：将汉字读音和形状结合起来考虑的编码，它吸取了字音编码和字形编码的优点，使编码规则简化、重码减少，但不易掌握。

2. 字符集与汉字内码

汉字通过输入码从键盘或通过语音识别从麦克风，或通过联机手写或印刷体文字扫描输入等各种手段被输入到计算机内部后，就按照一种称为“内码”的编码形式在系统中进行存储、查找、传送等处理。对于西文字符数据，它的内码就是ASCII码。对于汉字内码的选择，我们必须考虑以下几个因素：

(1) 不能有二义性，即不能和ASCII码有相同的编码。

(2) 要与汉字在字库中的位置有关系，以便于汉字的处理、查找。

(3) 编码应尽量短。

为了适应计算机处理汉字信息的需要，1981年我国颁布了《信息交换用汉字编码字符集·基本集》(GB2312—80)。该标准选出6763个常用汉字，为每个汉字规定了标准代码，以供汉字信息在不同计算机系统之间交换使用。这个标准称为国标码，又称国标交换码。

GB2312国标字符集由三部分组成：第一部分是字母、数字和各种符号，包括英文、俄文、日文平假名与片假名、罗马字母、汉语拼音等共687个；第二部分为一级常用汉字，共3755个，按汉语拼音排列；第三部分为二级常用字，共3008个，因为不太常用，所以按偏旁部首排列。

GB2312国标字符集中为任意一个字符(汉字或其他字符)规定了一个唯一的二进制代码。码表由94行(十进制编号0～93行)、94列(十进制编号0～93列)组成，行号称为区号，列号称为位号。每一个汉字或符号在码表中都有各自的位置，因此各有一个唯一的位置编码，该编码用字符所在的区号及位号的二进制代码表示，7位区号在左、7位位号在右，共14位，这14位代码就叫汉字的“区位码”。因此区位码指出了汉字在码表中的位置。

汉字的区位码并不是其国标码(即国标交换码)。由于信息传输的原因，每个汉字的区号和位号必须各自加上32(即十六进制的20H)，这样区号和位号各自加上32后的相应的二进制代码才是它的“国标码”，因此在“国标码”中区号和位号还是各自占7位。在计算机内部，为了处理与存储的方便，汉字国标码的前后各7位分别用一个字节来表示，所以共需两个字节才能表示一个汉字。因为计算机中的中西文信息是混合在一起进行处理的，所以汉字信息如不予以特别的标识，它与单字节的ASCII码就会混淆不清，无法识别。这就是前面给出的第一个要考虑的因素。为了解决这个问题，采用的方法之一，就是使表示汉字的两个字节的最高位(b_7)总等于“1”。这种双字节(16位)的汉字编码就是其中的一种汉字

“机内码”(即汉字内码)。例如,汉字“大”的区号是20,位号是83,因此区位码为14 53H(0001 0100 0101 0011B),国标码为34 73H(0011 0100 0111 0011B),前面的34H和字符“4”的ACSII码相同,后面的73H和字符“s”的ACSII码相同,将每个字节的最高位各设为“1”后,就得到其机内码B4 F3H(1011 0100 1111 0011B),这样就不会和ASCII码混淆了。应当注意,汉字的区位码和国标码是唯一的、标准的,而汉字内码可能随系统的不同而有差别。

随着亚洲地区计算机应用的普及与深入,汉字字符集及其编码还在发展。国际标准ISO/IEC 10646提出了一种包括全世界现代书面语言文字所使用的所有字符的标准编码,每个字符用4个字节(称为UCS—4)或两个字节(称为UCS—2)来编码。我国(包括香港、台湾地区)与日本、韩国联合制订了一个统一的汉字字符集(CJK编码),共收集了上述不同国家和地区的共约两万多汉字及符号,采用2字节(即UCS—2)编码,现已被批准为国家标准(GB13000)。美国微软公司在Windows操作系统(中文版)中也已采用了中西文统一编码,其中收集了中、日、韩三国常用的约两万汉字,称为Unicode(2字节编码),它与ISO/IEC 10646的UCS—2编码一致。

汉字输入码与汉字内码、汉字交换码完全是不同范畴的概念,不能把它们混淆起来。使用不同的输入编码方法输入同一个汉字时,在计算机内部得到的汉字内码是一样的。

3. 汉字的字模点阵码和轮廓描述

经过计算机处理后的汉字,如果需要在屏幕上显示出来或用打印机打印出来,则必须把汉字机内码转换成人们可以阅读的方块字形式。

每一个汉字的字形都必须预先存放在计算机内,一套汉字(例如GB2312国标汉字字符集)的所有字符的形状描述信息集合在一起称为字形信息库,简称字库(Font)。不同的字体(如宋体、仿宋、楷体、黑体等)对应着不同的字库。在输出每一个汉字时,计算机都要先到字库中去找到它的字形描述信息,然后把字形信息送到相应的设备输出。

汉字的字形主要有两种描述方法:字模点阵描述和轮廓描述。字模点阵描述是将字库中的各个汉字或其他字符的字形(即字模),用一个其元素由“0”和“1”组成的方阵(如16×16、24×24、32×32甚至更大)来表示,汉字或字符中有黑点的地方用“1”表示,空白处用“0”表示,我们把这种用来描述汉字字模的二进制点阵数据称为汉字的字模点阵码。汉字的轮廓描述方法比较复杂,它把汉字笔画的轮廓用一组直线和曲线来勾画,记下每一直线和曲线的数学描述公式。目前已有两类国际标准:Adobe Type1和True Type。这种用轮廓线描述字形的方式精度高,字形大小可以任意变化。

2.6 数据的宽度和存储

2.6.1 数据的宽度和单位

计算机内部任何信息都被表示成二进制编码形式。二进制数据的每一位(0或1)是组成二进制信息的最小单位,称为一个“比特”(bit),或称“位元”,简称“位”。比特是计算机中处理、存储和传输信息的最小单位。

每个西文字符需要用8个比特表示,而每个汉字需要用16个比特才能表示。因此,在

计算机内部，二进制信息的计量单位是“字节”(byte)，也称“位组”。一个字节等于8个比特。

计算机中运算和处理二进制信息时使用的单位除了比特和字节之外，还经常使用“字”(word)作为单位。必须注意，不同的计算机，字的长度和组成不完全相同，有的由两个字节组成，有的由4个、8个甚至16个字节组成。

在考察计算机性能时，一个很重要的指标就是机器的“字长”。平时所说的“某种机器是16位机或是32位机”中的16、32就是指字长。所谓“字长”通常是指CPU内部用于整数运算的数据通路的宽度。CPU内部数据通路是指CPU内部的数据流经的路径以及路径上的部件，主要是CPU内部进行数据运算、存储和传送的部件，这些部件的宽度基本上要一致，才能相互匹配。因此，“字长”等于CPU内部用于整数运算的运算器位数和通用寄存器宽度。

“字”和“字长”的概念不同，这一点请注意。“字”用来表示被处理信息的单位，用来度量各种数据类型的宽度。通常系统结构设计者必须考虑一台机器将提供哪些数据类型，每种数据类型提供哪几种宽度的数，这时就要给出一个基本的“字”的宽度。例如，Intel x86微处理器中把一个字定义为16位。所提供的数据类型中，就有单字宽度的无符号数和带符号整数(16位)、双字宽度的无符号数和带符号整数(32位)等。而“字长”表示进行数据运算、存储和传送的部件的宽度，它反映了计算机处理信息的一种能力。“字”和“字长”的长度可以一样，也可不一样。例如，在Intel微处理器中，从80386开始就至少都是32位机器了，即字长至少为32位，但其字的宽度都定义为16位，32位称为双字。

表示二进制信息存储容量时所用的单位要比字节或字大得多，主要有以下几种。

Kilo　1KB=2^{10}字节=1 024字节

Mega　1MB=2^{20}字节=1 048 576字节

Giga　1GB=2^{30}字节=1 073 741 824字节

Tera　1TB=2^{40}字节=1 099 511 627 776字节

Peta　1PB=2^{50}字节=1 125 899 906 842 624字节

Exa　1EB=2^{60}字节=1 152 921 504 606 846 976字节

Zetta　1ZB=2^{70}字节=1 180 591 620 717 411 303 424字节

Yotta　1YB=2^{80}字节=1 208 925 819 614 629 174 706 176字节

在描述距离、频率等数值时通常用10的幂次表示，因而在由时钟频率计算得到的总线带宽或外设数据传输率中，度量单位表示的是10的幂次。为区分这种差别，本书中用K表示1024，用k表示1000，而其他前缀字母均为大写，表示的大小由其上下文决定。

经常使用的带宽单位如下。

“比特/秒”(bps)　有时也写为bps

“千比特/秒”(kbps)　1kbps=10^3bps=1000bps

“兆比特/秒”(Mbps)　1Mbps=10^6bps=1000kbps

“吉比特/秒”(Gbps)　1Gbps=10^9bps=1000Mbps

“太比特/秒”(Tbps)　1Tbps=10^{12}bps=1000Gbps

由于程序需要对不同类型、不同长度的数据进行处理，所以，计算机中底层机器级的数据表示必须能够提供相应的支持。比如，需要提供不同长度的整数和不同长度的浮点数表

示，相应地需要有处理单字节、双字节、4字节、甚至是8字节整数的整数运算指令以及能够处理4字节、8字节浮点数的浮点数运算指令等。

C语言支持多种格式的整数和浮点数表示。数据类型char表示单个字节，能用来表示单个字符，也可用来表示8位整数。类型int之前可加上long和short，以提供不同长度的整数表示。表2.9给出了在典型的32位机器和64位的Compaq Alpha机器上C语言中数值数据类型的宽度。大多数32位机器使用“典型”方式。从表2.9可以看出，短整数为两个字节，普通int型整数为4个字节，而长整数的长度与机器字长的宽度相同。指针(例如，一个声明为类型char * 的变量)和长整数的宽度一样，也等于机器字长的宽度。一般机器都支持float和double两种类型的浮点数，分别对应IEEE 754单精度和双精度格式。

表2.9　C语言中数值数据类型的宽度

C声明	典型的32位机器	Compaq Alpha机器
char	1	1
short int	2	2
int	4	4
long int	4	8
char *	4	8
float	4	4
double	8	8

由此可见，同一类型的数据并不是所有机器都采用相同的数据宽度，分配的字节数随机器和编译器的不同而不同。

2.6.2　数据的存储和排列顺序

从前面介绍的各种类型数据的表示方法中可以了解到：任何信息在计算机中用二进制被编码以后，得到的都是一串0/1序列，每8位构成一个字节，不同的数据类型具有不同的字节宽度。在计算机中存储这些数据时，可有不同的存放顺序。数据从低位到高位可以按从左到右排列，也可以按从右到左排列。所以，用“最左位”(leftmost)和“最右位”(rightmost)来表示数据中的数位时会发生歧义。因此，一般用最低有效位(Least Significant Bit，LSB)和最高有效位(Most Significant Bit，MSB)来分别表示数的最低位和最高位。对于带符号数来说，最高位是符号位，所以MSB就是符号位。这样，不管数是从左往右排，还是从右往左排，只要明确MSB和LSB的位置，就可以明确表示数的符号和数值。例如，数“5”在32位机器上用int类型表示时的0/1序列为0000 0000 0000 0000 0000 0000 0000 0101，其中最左边的一位0是符号位，即MSB=0，最右边的“1”是数的最低有效位，即LSB=1。

如果以字节为一个排列基本单位，那么LSB表示最低有效字节(Least Significant Byte)，MSB表示最高有效字节(Most Significant Byte)。现代计算机基本上都采用字节编址方式，即对存储空间的存储单元进行编号时，每个地址编号中存放一个字节。计算机中许多类型的数据由多个字节组成，例如，int和float型数据占用4个字节，double型数据占用

8 个字节等，而程序中对每个数据只给定一个地址。例如，在一个按字节编址的计算机中，假定 int 型变量 i 的地址为 08 00H，i 的机器数为 01 23 45 67H，这 4 个字节 01H、23H、45H、67H 应该各有一个内存地址，那么，地址 08 00H 对应 4 个字节中的哪个字节的地址呢？这就是字节排列顺序问题。

在所有计算机中，多字节数据都被存放在连续的字节序列中。根据数据中各字节在连续字节序列中的排列顺序的不同，可有两种排列方式：大端（big endian）和小端（little endian），如图 2.5 所示。

		08 00H	08 01H	08 02H	08 03H	
大端方式	…	01H	23H	45H	67H	…
		08 00H	08 01H	08 02H	08 03H	
小端方式	…	67H	45H	23H	01H	…

图 2.5　大端方式和小端方式

大端（big endian）方式将数据的最高有效字节 MSB 存放在低地址单元中，将最低有效字节 LSB 存放在高地址单元中，即数据的地址就是 MSB 所在的地址。IBM 360/370、Motorola 68k、MIPS、Sparc、HP PA 等机器都采用大端方式。

小端（little endian）方式将数据的最高有效字节 MSB 存放在高地址中，将最低有效字节 LSB 存放在低地址中，即数据的地址就是 LSB 所在的地址。Intel 80x86、DEC VAX 等都采用小端方式。

有些微处理器芯片，如 Alpha 和 Motorola 的 Power PC，能够运行在任意一种模式，只要在芯片加电启动时选择确定采用大端还是小端模式即可。每个计算机系统内部的数据排列顺序都是一致的，但在系统间通信时可能会发生问题。在排列顺序不同的系统之间进行数据通信时，需要进行顺序转换。网络应用程序员必须遵守字节顺序的有关规定，以确保发送方机器将它的内部表示格式转换为网络标准，而接收方机器则将网络标准转换为自己的内部表示格式。

此外，像音频、视频和图像等文件格式或处理程序也都涉及到字节顺序问题。如 GIF、PC Paintbrush、Microsoft RTF 等采用小端方式，Adobe Photoshop、JPEG、MacPaint 等采用大端方式。

了解字节顺序的好处还在于调试底层机器级程序时，能够清楚每个数据的字节顺序，以便将一个机器数正确转换为真值。例如，以下是一个由反汇编器（反汇编是汇编的逆过程，即将指令代码转换为汇编表示）生成的一行针对某个处理器的机器级代码表示文本。

80483BD：01 05 64 94 04 08 add %eax，0x8049464

该文本行中，“80483BD”代表内存地址，是十六进制表示形式，“01 05 64 94 04 08”是指令代码，共 6 个字节，按顺序存放在地址 08 04 83 BDH 开始的 6 个连续内存单元中，add %eax，0x8049464 是指令的汇编形式。该指令的第二个操作数是一个立即数“0x8049464”，用十六进制数 08 04 94 64H 表示。指令执行时，直接从指令代码的后 4 个字节中取出该立即数，从指令代码中可看出，立即数在内存存放的字节序列为 64H 94H 04H 08H，

正好与操作数的字节序列相反。显然,该处理器采用的是小端方式。在阅读这种小端方式机器生成的机器级程序代码时,要记住字节是按照相反的顺序显示的。

例 2.27 以下是一段 C 程序,其中函数 show_int 和 show_float 分别用于显示 int 型和 float 型数据的值,show_pointer 用于显示指向 int 型数据的指针的值。显示的结果都用十六进制形式表示。

```
1 void test_show_bytes(int val)
2 {
3     int ival=val;
4     float fval= (float) ival;
5     int* pval= &ival;
6     show_int (ival);
7     show_float (fval);
8     show_pointer (pval);
9 }
```

上述程序在不同系统(Linux 和 Windows NT 运行于 Intel Pentium Ⅱ)上运行的结果见表 2.10。

表 2.10 程序在不同系统中的运行结果

系　　统	值	类　　型	字节(十六进制)
Linux	12345	int	39 30 00 00
Windows NT	12345	int	39 30 00 00
Sun	12345	int	00 00 30 39
Alpha	12345	int	39 30 00 00
Linux	12345.0	float	00 E4 40 46
Windows NT	12345.0	float	00 E4 40 46
Sun	12345.0	float	46 40 E4 00
Alpha	12345.0	float	00 E4 40 46
Linux	&ival	int *	3C FA FF BF
Windows NT	&ival	int *	1C FF 44 02
Sun	&ival	int *	EF FF FC E4
Alpha	&ival	int *	80 FC FF 1F 01 00 00 00

请回答下列问题。

(1) 十进制数 12 345 用 32 位补码整数和 32 位浮点数表示的结果各是什么?

(2) 十进制数 12 345 的整数表示和浮点数表示中存在一段相同位序列,标记出这段位序列,并说明为什么会相同。对一个负数来说,其整数表示和浮点数表示中是否也一定会出现一段相同的位序列? 为什么?

(3) Intel Pentium Ⅱ采用的是小端方式还是大端方式?

(4) Sun 和 Alpha 之间能否直接进行数据传送? 为什么?

(5) 在 Alpha 上,上述数据字节 30H 所存放的地址是什么?

解:(1) 十进制数 12 345 用 32 位补码整数表示为:0000 0000 0000 0000 0011 **0000**

0011 1001;用32位浮点数表示为:0100 0110 0**100 0000 1110 01**00 0000 0000。用十六进制表示分别为00 00 30 39H和46 40 E4 00H。

(2) 十进制数12 345的整数表示和浮点数表示中相同位序列为1 0000 0011 1001(见粗体部分)。因为对正数来说,原码和补码的编码相同,所以其定点整数和浮点数尾数的有效数位一样。12 345的有效数位是11 0000 0011 1001。有效数位在定点整数中位于低位数值部分,在浮点数的尾数中位于高位部分。因为尾数中有一个隐含的1,所以第一个有效数位在浮点数中不表示出来,因此,相同的位序列就是后面的13位。

因为IEEE 754浮点数的尾数用原码表示,而整数用补码表示,负数的原码和补码表示不同,所以,对某一个负数来说,其整数表示和浮点数表示中不一定会有相同的一段位序列。

(3) 从运行结果来看,Linux和Windows NT(运行在Intel Pentium Ⅱ)的存放方式与书写习惯顺序相反,故Intel Pentium Ⅱ采用的是小端方式。

(4) Sun和Alpha之间不能直接进行数据传送。因为它们采用了不同的存放方式,Sun是大端方式,这里Alpha设置的是小端方式。

(5) 在Alpha上数据字节30H存放在地址00 00 00 01 1F FF FC 81H中。因为从Alpha输出的int型指针结果来看,Alpha的主存地址占64位,30H是int型数据12 345的次低有效字节,小端方式下数据地址取LSB的地址,所以30H存放的地址应该是数据地址随后的那个地址。根据小端方式下存放结果和书写习惯顺序相反的规律,可知数据12 345的地址是00 00 00 01 1F FF FC 80H,所以,随后的地址就是00 00 00 01 1F FF FC 81H。

2.7 数据校验码

数据在计算机内部进行计算、存取和传送过程中,由于元器件故障或噪音干扰等原因会出现差错。为了减少和避免这些错误,一方面要从计算机硬件本身的可靠性入手,在电路、电源、布线等各方面采取必要的措施,提高计算机的抗干扰能力;另一方面要采取相应的数据检错和校正措施,自动地发现并纠正错误。

目前为止提出的数据校验方法大多采用一种“冗余校验”的思想,即除原数据信息外,还增加若干位编码,这些新增的代码被称为校验位。图2.6给出了一般情况下的处理过程。

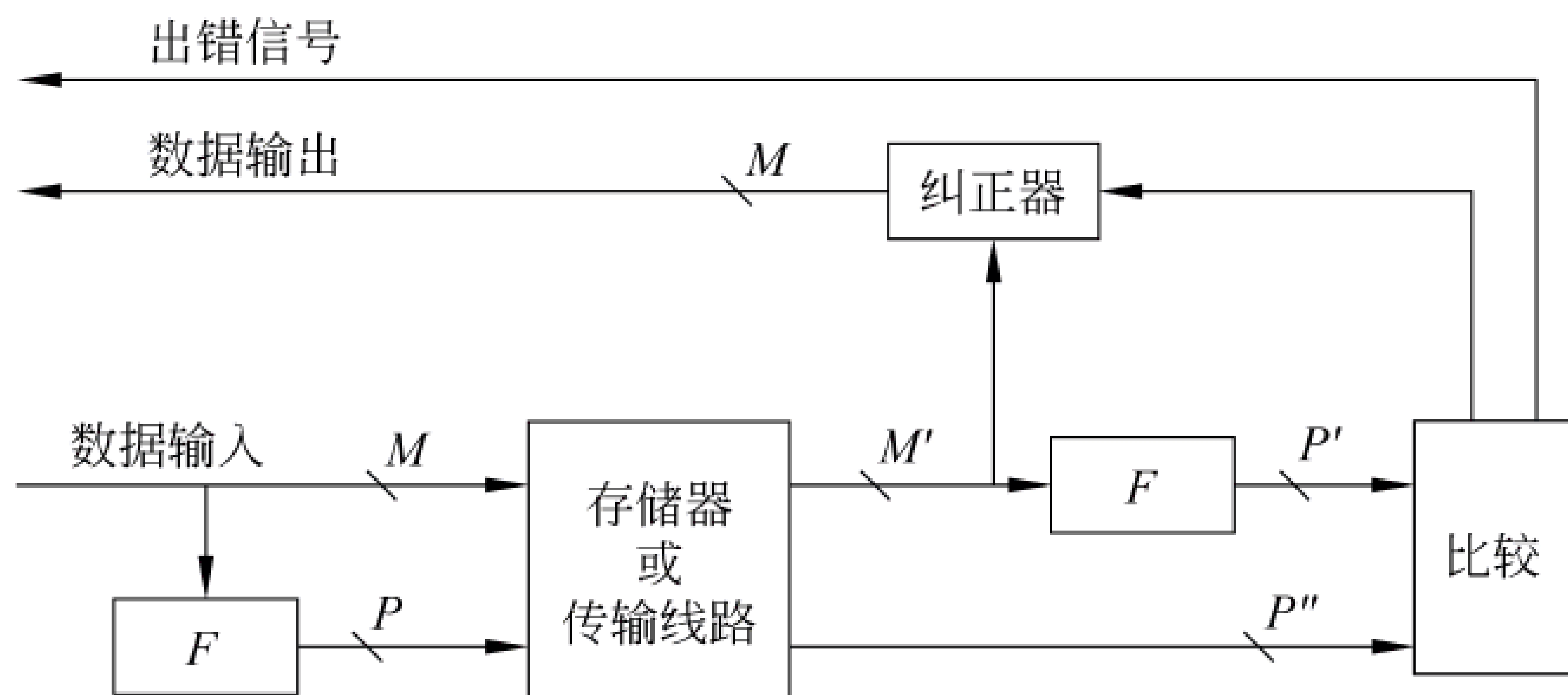


图2.6 数据校验过程示意图

当数据被存入存储器或从源部件传输时，对数据 M 进行某种运算（用函数 F 来表示），以产生相应的代码 $P=F(M)$，这里 P 就是校验位。这样原数据信息 M 和相应的校验位 P 一起被存储或传送。当数据被读出或传送到目标部件时，和数据信息一起被存储或传送的校验位也被得到，用于检错和纠错。假定读出后的数据为 M'，通过同样的运算 F 对 M' 也得到一个新的校验位 $P'=F(M')$，假定原来被存储的校验位 P 取出后其值为 P''，将校验位 P'' 与新生成的校验位 P' 进行某种比较，根据其比较结果确定是否发生了差错。比较的结果为以下三种情况之一。

（1）没有检测到错误，得到的数据位直接传送出去。

（2）检测到差错，并可以纠错。数据位和比较结果一起送入纠错器，然后将产生的正确的数据位传送出去。

（3）检测到错误，但无法确认哪位出错，因而不能进行纠错处理，此时，报告出错情况。

为了判断一种码制的冗余程度，并评估它的查错和纠错能力，引入了“码距”的概念。由若干位代码组成的一个字叫“码字”，将两个码字逐位比较，具有不同代码的位的个数叫做这两个码字间的“距离”，也称为“海明距离”。一种码制可能有若干个码字，而且，其中任意两个码字之间的距离可能不同，我们将各码字间的最小距离称为“码距”，它就是这个码制的距离。例如 8421 编码中，2(0010)和 3(0011)之间距离为 1，所以 8421 码制的码距为 1，记做 $d=1$。在数据校验码中，一个码字是指数据位和校验位按照某种规律排列得到的代码。

一般来说，合理地增加校验位、增大码距，就能提高检错/纠错的能力。例如，如果采用 4 位二进制编码 0000～1111 表示 16 种状态，则码距为 1，因为这组编码的任何一个 4 位码中出现一位或多位错误，都会变成另一个合法编码，因此，这种编码没有检错/纠错能力。如果采用 4 位二进制表示 8 个状态：0000、0011、0101、0110、1001、1010、1100、1111，其余的 4 位编码都是非法的。本来 8 个状态只要 3 位二进制就够了，现在增加一个冗余位，并使码距为 2，这样，这 8 个合法编码中如果某个编码发生了一位错误，那么就会变成一个非法编码，因而就能检测出来。一般地，要能检测 e 位错，编码的码距必须至少达到 $e+1$，这样的话，如果一个合法码字发生了 e 位错，就无法变为另一个合法码字，因而可以检测出来。对于纠错来说，如果出错位置确定，只要将其取反就能自动纠错。要能纠正 e 位错，编码的码距必须达到 $2e+1$。这样，如果一个合法码字中有 e 位出错，那么出错编码与原合法码字之间的距离为 e，而与其他合法码字之间的距离至少为 $e+1$，因而能够唯一确认出错编码的原合法码字，从而自动进行纠错处理。

例如，假定一个码制中有 4 个码字：00000 00000、00000 11111、11111 00000、11111 11111。显然，其码距为 5，所以能纠正两位错，但不能纠正 3 位及 3 位以上的错。例如，假定传送的是 00000 11111，传输后接收方收到的信息为 00000 01110，那么，根据 00000 01110 与 00000 11111 的距离最小，就能知道原来正确的编码为 00000 11111；假定传送的是 00000 00000，而接收到的是 00000 00111，那么，就无法确定原正确编码应该是 00000 00000，因为 00000 11111 与 00000 00111 之间的距离更小。

由此可见，码距与检错、纠错能力之间存在相应关系。当码距 $d\leqslant 4$ 时，关系如下：

（1）如果码距 d 为奇数，则能发现 $d-1$ 位错，或者能纠正 $(d-1)/2$ 位错。

（2）如果码距 d 为偶数，则能发现 $d/2$ 位错，并能纠正 $(d/2-1)$ 位错。

常用的数据校验码有奇偶校验码、海明校验码和循环冗余校验码。

2.7.1 奇偶校验码

最简单的一种数据校验方法是奇偶校验。奇偶校验法的基本思想是通过在原数据信息中增加一位奇校验位(或偶校验位),然后将原数据和得到的奇(偶)校验位一起进行存储或传送,对从存储器取出的数据或传送到目标部件的数据以及奇(偶)校验位,再进行一次编码,求出新的奇(偶)校验位,最后根据得到的这个新的校验位的值,确定是否发生了错误。

奇偶校验码的实现原理如下:假设将数据 $B=b_{n-1}b_{n-2}\cdots b_1b_0$ 从源部件传送至目标部件。在目标部件接收到的数据为 $B'=b'_{n-1}b'_{n-2}\cdots b'_1b'_0$。为了判断数据 B 在传送中是否发生了错误,可以按照如下步骤来判断。

第一步:在源部件求出奇(偶)校验位 P。

若采用奇校验位,则 $P=b_{n-1}\oplus b_{n-2}\oplus\cdots\oplus b_1\oplus b_0\oplus 1$。即当 B 有奇数个 1 时 P 取 0,否则,P 取 1。

若采用偶校验位,则 $P=b_{n-1}\oplus b_{n-2}\oplus\cdots\oplus b_1\oplus b_0$。

举例:若传送的是字符 A:1000001,则在前面增加奇校验位后的编码为 1 1000001,而加上偶校验位后的编码为 0 1000001。

第二步:在目标部件求出奇(偶)校验位 P'。

若采用奇校验位,则 $P'=b'_{n-1}\oplus b'_{n-2}\oplus\cdots\oplus b'_1\oplus b'_0\oplus 1$。

若采用偶校验位,则 $P'=b'_{n-1}\oplus b'_{n-2}\oplus\cdots\oplus b'_1\oplus b'_0$。

第三步:计算最终的校验位 P^*,并根据其值判断有无奇偶错。

P 与 B 是一起从源部件传到目标部件的,假定 P 在目标部件接收到的值为 P'',则 $P^*=P'\oplus P''$。

(1) 若 $P^*=1$,则表示有奇数位错。

(2) 若 $P^*=0$,则表示正确或有偶数位错。

在奇偶校验码中,若两个数据中有奇数位不同,则它们相应的校验位就不同;若有偶数位不同,则虽校验位相同,但至少有两位数据位不同,因而任意两个码字之间至少有两位不同,所以码距 $d=2$。根据码距和检/纠错能力的关系可知,它只能发现奇数位出错,不能发现偶数位出错,而且也不能确定发生错误的位置,不具有纠错能力。但奇偶校验法所用的开销小,它常被用于存储器读写检查或按字节传输过程中的数据校验。因为一字节长的代码中一位出错的概率相对较大,两位以上出错则很少,所以奇偶校验码用于校验一字节长的代码还是有效的。

2.7.2 海明校验码

前面所述的奇偶校验码是对整个数据编码生成一位校验位。因此这种校验码检错能力差,并且没有纠错能力。海明校验码由 Richard Hamming 于 1950 年提出,其主要思想是:将数据按某种规律分成若干组,对每组进行相应的奇偶检测,以提供多位校验信息,从而可对错误位置进行定位,并将其纠正。海明校验码实质上就是一种多重奇偶校验码。

海明校验码的处理过程与图 2.6 给出的一般过程一样。最终进行比较时,按位进行异或操作,根据异或操作的结果,确定是否发生了差错。这种异或操作所得到的结果被称为故障字(syndrome word)。显然,校验位和故障字的位数是相同的。

1. 校验位的位数的确定

要实现对某个数据发生的错误进行定位，该数据的校验位至少应有几位呢？假定该数据的位数为 n，校验位为 k 位，则故障字的位数也为 k 位。那么 k 位的故障字所能表示的状态最多是 2^k 种，每种状态可用来说明一种出错情况。对于最多只有一位错的情况，其结果可能是无错或 n 位数据中某一位出错或 k 位校验码中某一位出错。因此，共有 $1+n+k$ 种情况，所以，要能对一位错的所有结果进行正确表示，则 n 和 k 必须满足下列关系：

$$2^k \geqslant 1+n+k, \quad 即 \quad 2^k-1 \geqslant n+k$$

表 2.11 给出了数据位和校验位的对应位数，以及增加的校验位占数据位的百分比。

表 2.11 数据位和校验位间的位数关系

数据位数	单 纠 错		单纠错/双检错	
	校验位数	增加的百分比	校验位数	增加的百分比
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.25	10	3.91

从表 2.11 中可以看出，当数据有 8 位时，校验位和故障字都应有 4 位。4 位的故障字最多可以表示 16 种状态，而单个位出错情况最多只有 12 种可能(8 个数据位和 4 个校验位)，再加上无错的情况，一共有 13 种。所以，用 16 种状态表示 13 种情况应是足够了。

2. 分组方式的确定

数据位和校验位是一起被存储的，通过将它们中的各位按某种方式排列为一个$(n+k)$位的码字，将该码字中每一位的出错位置与故障字的数值建立关系，这样就可通过故障字的值很快确定是该码字中的哪一位发生了错误，从而将其取反来进行纠正。

根据上述基本思想，我们按以下规则来解释各故障字的值。

(1) 如果故障字各位全部是 0，则表示没有发生错误。

(2) 如果故障字中有且仅有一位为 1，则表示校验位中有一位出错，不需要纠正。

(3) 如果故障字中多位为 1，则表示有一个数据位出错，其在码字中的出错位置由故障字的数值来确定。纠正时只要将出错位取反即可。

为了介绍海明校验码的原理，这里以 8 位数据进行单个位的检错/纠错为例说明。假定一个 8 位数据 $M=M_8M_7M_6M_5M_4M_3M_2M_1$，其相应的 4 位校验位为 $P=P_4P_3P_2P_1$。根据上述规则将数据 M 和校验位 P 按照一定的规律排到一个 12 位的码字中。根据上述第一个规则，故障字为 0000 时，表示无错，因此没有和位置号 0000 对应的出错情况，所以位置号从 0001 开始。根据第二个规则，故障字中有且仅有一位为 1 时，表示校验位中有一位出错，此时，故障字只可能是 0001、0010、0100、1000 这 4 种情况，将这 4 种状态分别代表校验位中第 P_1、P_2、P_3、P_4位发生错误的情况，因此，校验位 P_1、P_2、P_3、P_4应分别位于码字的第 0001(1)、0010(2)、0100(4)、1000(8)位。根据最后一个规则，将其他多位为 1 的故障字依次表示数据

位 $M_1 \sim M_8$ 发生错误的情况。因此，数据位 $M_1 \sim M_8$ 应分别位于码字的第 0011(3)、0101(5)、0110(6)、0111(7)、1001(9)、1010(10)、1011(11)、1100(12)位。即码字的排列为：

$$M_8 M_7 M_6 M_5 P_4 M_4 M_3 M_2 P_3 M_1 P_2 P_1$$

这样得到故障字 $S = S_4 S_3 S_2 S_1$ 的各个状态和出错情况的对应关系，如表 2.12 所示。因为故障字的值决定了哪位出错，所以，某位出错一定会影响与之相对应的故障字中为 1 的位所在组的奇偶性。例如，若 M_1 出错，则对应故障字为 0011，因此一定会改变 S_1 和 S_2 所在分组的奇偶性，故 M_1 应同时被分到与 S_1 对应的第 1 组和与 S_2 对应的第 2 组。同理，P_1 对应故障字 0001，故 P_1 应被分到与 S_1 对应的第 1 组；M_8 对应故障字 1100，故应分到与 S_3 对应的第 3 组和与 S_4 对应的第 4 组。表 2.12 中根据这种对应关系对数据进行了分组。每组有一个奇偶校验位 P_i。若某组 i 中的一位发生错误，那么该组对应的那些位的奇偶性发生变化，可根据该组对应的故障位 S_i 的值是否为 1，来判断奇偶性是否发生了变化，从而确定该组是否发生了一位错误。

表 2.12　故障字和出错情况的对应关系

分组 \ 序号	1	2	3	4	5	6	7	8	9	10	11	12	故障字	正确	出错位
含义	P_1	P_2	M_1	P_3	M_2	M_3	M_4	P_4	M_5	M_6	M_7	M_8			1 2 3 4 5 6 7 8 9 10 11 12
第 4 组								√	√	√	√	√	S_4	0	0 0 0 0 0 0 0 1 1 1 1 1
第 3 组				√	√	√	√					√	S_3	0	0 0 0 1 1 1 1 0 0 0 0 1
第 2 组		√	√			√	√			√	√		S_2	0	0 1 1 0 0 1 1 0 0 1 1 0
第 1 组	√		√		√		√		√		√		S_1	0	1 0 1 0 1 0 1 0 1 0 1 0

从表 2.12 中可以看出，每一个数据位至少要参与两组奇偶校验位的生成，如 M_5 与第 1 组(P_1)和第 4 组(P_4)有关，M_7 与第 1 组(P_1)、第 2 组(P_2)和第 4 组(P_4)有关。

3. 校验位的生成和检错、纠错

分组完成后，就可对每组采用相应的奇(偶)校验，以得到相应的一个校验位。假定采用偶校验(即取校验位 P_i，使对应组中有偶数个 1)，则得到校验位与数据位之间存在如下关系：

$$P_1 = M_1 \oplus M_2 \oplus M_4 \oplus M_5 \oplus M_7$$
$$P_2 = M_1 \oplus M_3 \oplus M_4 \oplus M_6 \oplus M_7$$
$$P_3 = M_2 \oplus M_3 \oplus M_4 \oplus M_8$$
$$P_4 = M_5 \oplus M_6 \oplus M_7 \oplus M_8$$

根据上面的公式，可以求出每一组对应的校验位 $P_i(i=1,2,3,4)$。数据 M 和校验位 P 一起被存储。读出后的数据 M' 通过上述同样的公式得到新的校验位 P'，然后将读出后的校验位 P'' 与新生成的校验位 P' 按位进行异或操作，得到故障字 $S = S_4 S_3 S_2 S_1$，根据 S 的值可以确定是否发生了错误，并且在发生错误时能确定是校验位发生错误还是哪个数据位发生了错误。

下面举例说明具体的检错、纠错过程。

例 2.28　假定一个 8 位数据 M 为 $M_8 M_7 M_6 M_5 M_4 M_3 M_2 M_1 = 01101010$，其对应的校验位为 P，M 和 P 被存储或经传送后得到的新数据和新校验码为 M' 和 P''。分别求出以下 3

种情况下的故障字并验证其正确性。

(1) $M'=01101010$，$P''=0011$。

(2) $M'=01111010$，$P''=0011$。

(3) $M'=01101010$，$P''=1011$。

解：根据上述公式求出数据 M 的校验位 P 中相应各位。

$$P_1 = M_1 \oplus M_2 \oplus M_4 \oplus M_5 \oplus M_7 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 1$$
$$P_2 = M_1 \oplus M_3 \oplus M_4 \oplus M_6 \oplus M_7 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$
$$P_3 = M_2 \oplus M_3 \oplus M_4 \oplus M_8 = 1 \oplus 0 \oplus 1 \oplus 0 = 0$$
$$P_4 = M_5 \oplus M_6 \oplus M_7 \oplus M_8 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$

(1) 当 $M'=01101010$，$P''=0011$ 时，因为 $M'=M$，说明数据无错，故障字 S 应为 0。验证如下：因 $M'=M$，故 $P'=P$，同时 $P''=P$，因此故障字 $S=P''\oplus P'=P\oplus P=0000$。

(2) 当 $M'=01111010$，$P''=0011$ 时，说明数据第 5 位(M_5)错，故障字 S 应该为 9。验证如下。

对 M' 生成新的校验位 P' 为：

$$P'_1 = M'_1 \oplus M'_2 \oplus M'_4 \oplus M'_5 \oplus M'_7 = 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 = 0$$
$$P'_2 = M'_1 \oplus M'_3 \oplus M'_4 \oplus M'_6 \oplus M'_7 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$
$$P'_3 = M'_2 \oplus M'_3 \oplus M'_4 \oplus M'_8 = 1 \oplus 0 \oplus 1 \oplus 0 = 0$$
$$P'_4 = M'_5 \oplus M'_6 \oplus M'_7 \oplus M'_8 = 1 \oplus 1 \oplus 1 \oplus 0 = 1$$

因此，故障字 S 为：

$$S_1 = P'_1 \oplus P''_1 = 0 \oplus 1 = 1$$
$$S_2 = P'_2 \oplus P''_2 = 1 \oplus 1 = 0$$
$$S_3 = P'_3 \oplus P''_3 = 0 \oplus 0 = 0$$
$$S_4 = P'_4 \oplus P''_4 = 1 \oplus 0 = 1$$

根据故障字的值 1001，可以判断出发生错误的位是在 12 位码字的第 1001 位(即第 9 位)，在这一位上排列的是数据位 M_5，验证正确。纠错时，只要将码字的第 9 位(即第 5 个数据位)取反即可。

(3) 当 $M'=01101010$，$P''=1011$ 时，说明数据 M 无错，校验位 P_4 错，故障位 S 应为 8。验证如下。

因为 $M'=M$，所以 $P'=P$，故障位 S 为：

$$S_1 = P'_1 \oplus P''_1 = 1 \oplus 1 = 0$$
$$S_2 = P'_2 \oplus P''_2 = 1 \oplus 1 = 0$$
$$S_3 = P'_3 \oplus P''_3 = 0 \oplus 0 = 0$$
$$S_4 = P'_4 \oplus P''_4 = 0 \oplus 1 = 1$$

根据故障字的值 1000，可以判断出发生错误的位是在 12 位码字的第 1000 位(即第 8 位)，在这一位上排列的是校验位 P_4，验证正确。这种情况下不需纠错。

从上述对数据位数 $n=8$、校验位数 $k=4$ 的分组情况来看，如果两个数据有一位不同，那么由于该位至少要参与两组校验位的生成，因而至少会引起两个校验位的不同，再加上数

据位本身一位的不同，所以其码距 $d=3$。根据码距与检错、纠错能力的关系，可知这种码制若用来判别有无错误，则能发现两位错；若用来纠错，则只能对单个位出错进行定位和纠错，因此被称为单纠错码(SEC)。

若校验码同时具有发现两位错和纠正一位错的能力，则称为单纠错和双检错码(SEC-DED)，简称“纠一检二”码。若要使上述介绍的单纠错码成为 SEC-DED 码，则码距需扩大到 $d=4$。为此，还需要增加一位校验位 P_5，可将 P_5 排列在码字的最前面，加入 P_5 后的排列顺序为：

$$P_5 M_8 M_7 M_6 M_5 P_4 M_4 M_3 M_2 P_3 M_1 P_2 P_1$$

此外，还必须使得数据中的每一位都参与至少 3 个校验位的生成，从表 2.12 中可看出除了 M_4 和 M_7 参与了 3 个校验位的生成外，其余位都只参与了两个校验位的生成，因此可将这些位加入到 P_5 所在的组中，得到以下公式：

$$P_5 = M_1 \oplus M_2 \oplus M_3 \oplus M_5 \oplus M_6 \oplus M_8$$

这样，当任意一个数据位发生错误时，必将引起 3 个校验位发生变化，所以码距为 4。

引入 P_5 后，故障字 S 也增加了一位，即 $S=S_5S_4S_3S_2S_1$，按照类似的方法可以求出故障字 S 中各位的值，根据 $S_5S_4S_3S_2S_1$ 的取值情况，即可按照如下规则发现两位错并纠正一位错。

(1) 当 $S_5S_4S_3S_2S_1$ 为 00000 时，表明无错。

(2) 当 $S_5S_4S_3S_2S_1$ 中仅一位不为 0 时，表明由 S 指定位置上的那个校验位发生了错误，或是在数据和校验位中有 3 位同时出错，但后面这种可能性非常小，所以一般认为就是发生了前一种情况。

(3) 当 $S_5S_4S_3S_2S_1$ 中有两位不为 0 时，表明数据和校验位中有两位同时出错，此时只能发现这种错误，但无法确定是哪两位错。

(4) 当 $S_5S_4S_3S_2S_1$ 中有 3 位不为 0 时，表明有一个数据位发生了错误，或是 3 个校验位同时出错，但后面这种可能性非常小，所以一般认为就是发生了前一种情况。此时，出错的位置由 $S_4S_3S_2S_1$ 的数值指定。

(5) 当 $S_5S_4S_3S_2S_1$ 中有 4 位或 5 位都不为 0 时，表明出错情况严重，系统可能出现故障，应检查系统硬件的正确性。

*2.7.3 循环冗余校验码

循环冗余校验码(Cyclic Redundancy Check，CRC 码)，是一种具有较强检错、纠错能力的校验码，常用于外存储器的数据校验，在计算机通信中，也被广泛采用。在数据传输中，奇偶校验码是在每个字符信息后增加一位奇偶校验位来进行数据校验的，这样对大批量传输的数据进行校验时，会增加大量的额外开销，尤其是在网络通信中，传输的数据信息都是二进制比特流，因而没有必要将数据再分解成一个个字符，这样也就无法采用奇偶校验码，因此，通常采用 CRC 码进行校验。

前面所介绍的奇偶校验码和海明校验码都是以奇偶检测为手段的，而循环冗余校验码则是通过某种数学运算来建立数据和校验位之间的约定关系。因为 CRC 码的编码原理复杂，这里仅对其编码方式和实现过程作简单介绍，而不详细进行数学推导。

1. CRC 码的检错方法

假设要进行校验的数据信息 $M(x)$ 为一个 n 位的二进制数据，将 $M(x)$ 左移 k 位后，用一个约定的生成多项式 $G(x)$ 相除，$G(x)$ 是一个 $k+1$ 位的二进制数，相除后得到的 k 位余数就是校验位。这些校验位拼接到 $M(x)$ 的 n 位数据后面，形成一个 $n+k$ 位的代码，称这个代码为循环冗余校验码（CRC 码），也称 $(n+k, n)$ 码，如图 2.7 所示。一个 CRC 码一定能被生成多项式整除，所以当数据和校验位一起送到接收端后，只要将接收到的数据和校验位用同样的生成多项式相除，如果正好除尽，表明没有发生错误；若除不尽，则表明某些数据位发生了错误。

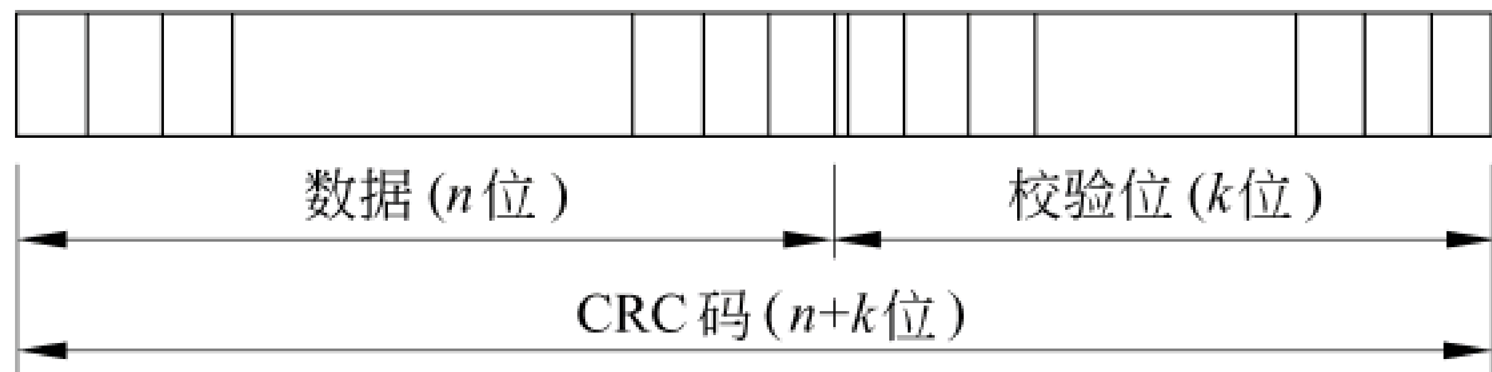


图 2.7 CRC 码的组成

2. 校验位的生成

下面用一个例子来说明校验位的生成过程。假设要传送的数据信息为 100011，数据信息位数 $n=6$，报文多项式为 $M(x)=x^5+x+1$。若约定的生成多项式为 $G(x)=x^3+1$，即生成多项式位数为 4，则校验位位数 $k=3$。生成校验位时，用 $x^3M(x)$ 去除以 $G(x)$，相除时采用“模 2 运算”的多项式除法。模 2 运算时不考虑加法进位和减法借位。进行模 2 除法时，上商的原则是当部分余数首位是 1 时上商为 1，反之上商为 0。然后按模 2 相减原则求得最高位后面几位的余数。这样当被除数逐步除完时，最后的余数位数比除数少一位。得到的余数就是校验码，此例中最终的余数有 3 位。

图 2.8 说明利用“模 2”多项式除法计算 $x^3M(x) \div G(x)$ 的过程。

$$x^3M(x) \div G(x) = (x^8+x^4+x^3) \div (x^3+1) = x^2+x+1$$

图 2.8(a)计算出的校验位为 111，CRC 码为 100011 111。如果要检查 CRC 码，可将 CRC 码用同一个多项式相除，若余数为 0，则说明无错；若余数不为 0，则说明有错。例如：

```
          100111
     ___________
1001 ) 100011000
       1001
       ----
        0011
        0000
        ----
         0111
         0000
         ----
          1110
          1001
          ----
           1110
           1001
           ----
            1110
            1001
            ----
             111
```

(a) 计算校验位

```
          100111
     ___________
1001 ) 100011111
       1001
       ----
        0011
        0000
        ----
         0111
         0000
         ----
          1111
          1001
          ----
           1101
           1001
           ----
            1001
            1001
            ----
             000
```

(b) 余数为 0，数据正确

```
          101110
     ___________
1001 ) 101011111
       1001
       ----
        0111
        0000
        ----
         1111
         1001
         ----
          1101
          1001
          ----
           1001
           1001
           ----
            0001
            0000
            ----
             001
```

(c) 余数不为 0，数据不正确

图 2.8 CRC 的校验位计算及验证

若接收方的 CRC 码与发送方一致，即为 100011 111 时，用同一个多项式相除后余数为 0，如图 2.8(b)所示；若接收方的 CRC 码有一位出错而变为 101011 111 时，用同一个多项式相除后余数不为 0，如图 2.8(c)所示。

3. CRC 码的纠错

当接收方将收到的 CRC 码用约定的生成多项式 $G(x)$ 去除，发现余数不为 0 时，需要判断出错的位置。不同的出错位置其余数不同，而且对于不同的码字，在确定的码制与生成多项式下，只要出错位置相同，则余数一定相同。例如，表 2.13 给出了(7,4)循环码中生成多项式 $G(x)=1011$ 时出错位置与余数的关系。表中给出两种不同的码字，其中加粗的是出错位，可以看出其出错位置与余数的关系是相同的。对于其他码制或选用其他生成多项式，出错位置和余数的关系可能与表 2.13 所示的不同。

表 2.13 码字、余数和出错位的关系

	码字举例														余数	出错位
	D_1	D_2	D_3	D_4	P_1	P_2	P_3	D_1	D_2	D_3	D_4	P_1	P_2	P_3		
正确	1	0	1	0	0	1	1	1	0	1	1	0	0	0	000	无
错误	1	0	1	0	0	1	**0**	1	0	1	1	0	0	**1**	001	7
	1	0	1	0	0	**0**	1	1	0	1	1	0	**1**	0	010	6
	1	0	1	0	**1**	1	1	1	0	1	1	**1**	0	0	100	5
	1	0	1	**1**	0	1	1	1	0	1	**0**	0	0	0	011	4
	1	0	**0**	0	0	1	1	1	0	**0**	1	0	0	0	110	3
	1	**1**	1	0	0	1	1	1	**1**	1	1	0	0	0	111	2
	0	0	1	0	0	1	1	**0**	0	1	1	0	0	0	101	1

如果 CRC 码中有一位出错，用特定的 $G(x)$ 进行模 2 除，则会得到一个不为 0 的余数。若对余数补 0 后继续除下去，则会出现一个有趣的现象：各次余数将会按照一个特定的顺序循环。例如在表 2.13 所示的例子中，若将第 7 位出错时对应的余数 001 后面补 0，继续再除一次，则会得到新余数 010，在 010 后补 0，继续再除一次，则会得到下一个余数 100，如此继续下去，依次得到 011，110，111，…，反复循环。这是为什么称为“循环”冗余码的由来。利用这种特点，能方便地对出错码字进行纠错，所用硬件开销小。在大批量数据传输校验中，能有效地降低硬件成本。

4. 生成多项式的选取

并不是任何一个多项式都能作为生成多项式。从查错和纠错的要求来看，选取的一个生成多项式应满足以下几个条件：

(1) 任何一位发生错误时，都应使余数不为 0。

(2) 不同位发生错误时，余数应该不同。

(3) 对余数作模 2 除时，应使余数循环。

将这些条件用数学方式描述起来比较复杂，对一个 $(n+k,n)$ 码来说，可将 $(x^{n+k}-1)$ 按模 2 运算分解为若干质因子，根据所要求的码距，选取其中的因式或若干因式的乘积作为生成多项式。例如，若要求对一个 $(7,n)$ 码选取相应的生成多项式，可以按上述方法对 (x^7-1) 分解质因子。

$$x^7-1=(x+1)(x^3+x+1)(x^3+x^2+1)\quad (\text{模 2 运算})$$

若选择 $G(x)=x+1=11$，则可构成(7,6)码，只能发现一位错。若选择 $G(x)=x^3+x+1=1011$ 或 $G(x)=x^3+x^2+1=1101$，则可构成(7,4)码，能纠正一位错或发现两位错。若选择 $G(x)=(x+1)(x^3+x+1)=11101$，则可构成(7,3)码，能纠正一位错并发现两位错。

下面是几种常用的生成多项式。

CRC-CCITT：$G(x)=x^{16}+x^{12}+x^5+1$

CRC-16：$G(x)=x^{16}+x^{15}+x^2+1$

CRC-12：$G(x)=x^{12}+x^{11}+x^3+x^2+x+1$

CRC-32：$G(x)=x^{32}+x^{26}+x^{23}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

在网络通信中，通常数据位数 n 相当大，由几千个二进位构成一帧数据，因此，通常使用 CRC 码来检测错误，发现错误则告知发送方要求重发，不用 CRC 来纠正错误。因此，只要在接收方用约定的生成多项式进行模 2 除后判断余数是否为 0 就行了。

2.8 本章小结

本章主要介绍在计算机内部数据的机器级表示、数据的宽度和存储排列顺序以及数据检错和纠错方法。有关数据的表示主要包括：真值和机器数的概念，无符号数的表示，带符号整数的表示，浮点数的表示（包括浮点数的形式、浮点数的规格化、浮点数的表示范围、IEEE 754 标准），十进制数的二进制编码表示以及逻辑值、西文字符和汉字字符等非数值数据的机内表示。

本章内容具体总结如下。

- 数据的表示：有数值数据与非数值数据两大类。
  - 数值数据：在数轴上有对应点、能比较大小的数，有二进制和十进制两种形式。
    - 用二进制表示的数：直接用二进制表示。
      无符号数：正整数，用来表示地址等。
      带符号整数：表示整数，一般用补码表示。
      浮点数：表示实数，大多用 IEEE 754 标准表示。
    - 用十进制表示的数：用二进制进行编码，称为 BCD 码。一般用 8421 码表示。
  - 非数值数据：在数轴上没有对应点的数据。
    - 逻辑值：只有两个状态取值，按位进行运算。
    - 西文字符：多采用 7 位 ASCII 码表示。
    - 汉字字符：有输入码、内码和字模码，内码大多占两个字节。
- 数据的宽度：通常以字节(Byte)为基本单位表示，数据长度单位（如 MB、GB、TB 等）在表示数据容量和带宽等不同对象时所代表的大小不同。
- 数据的排列：有大端和小端两种排列方式。
  - 大端方式：以 MSB 所在地址为数据地址，即给定地址处存放最高有效字节。
  - 小端方式：以 LSB 所在地址为数据地址，即给定地址处存放最低有效字节。

- 常用数据校验方式有以下 3 种。
  - 奇偶校验：根据数据的奇偶性变化来检错，只能检测奇数位错。
  - 海明校验：分组奇偶校验，SEC 只能纠正一位错，SEC-DED 可纠正一位错并检测两位错。
  - 循环冗余码(CRC)校验：通过某种数学运算在数据和校验位之间建立约定关系，它可以对较长数据块进行校验而不增加校验位开销，因此，主要用于对大批量数据的存储或传输校验。

习　题　2

1. 给出以下概念的解释说明。

(1) 真值　(2) 机器数　(3) 数值数据　(4) 非数值数据　(5) 无符号整数
(6) 带符号整数　(7) 定点数　(8) 原码　(9) 反码　(10) 补码
(11) 变形补码　(12)浮点数　(13) 尾数　(14) 阶码　(15) 移码
(16) 溢出　(17) 下溢　(18) 上溢　(19) 规格化数　(20) 左规
(21) 右规　(22) 非规格化数　(23) 机器零　(24) 非数(NaN)　(25) BCD 码
(26) 逻辑数　(27) ASCII 码　(28) 汉字输入码　(29) 汉字内码　(30) 机器字长
(31) 大端方式　(32) 小端方式　(33) 字地址　(34) 边界对齐　(35) 检错
(36) 纠错　(37) 码距　(38) 奇偶校验　(39) 海明码
(40) 循环冗余校验(CRC)　(41) 最高有效位(MSB)
(42) 最高有效字节(MSB)　(43) 最低有效位(LSB)
(44) 最低有效字节(LSB)

2. 简单回答下列问题。

(1) 为什么计算机内部采用二进制表示信息？既然计算机内部所有信息都用二进制表示，为什么还要用到十六进制或八进制数？

(2) 常用的定点数编码方式有哪几种？通常它们各自用来表示什么？

(3) 为什么现代计算机中大多用补码表示带符号整数？

(4) 在浮点数的基数和总位数一定的情况下，浮点数的表示范围和精度分别由什么决定？两者如何相互制约？

(5) 为什么要对浮点数进行规格化？有哪两种规格化操作？

(6) 为什么有些计算机中除了用二进制外还用 BCD 码来表示数值数据？

(7) 为什么计算机处理汉字时会涉及到不同的编码(如输入码、内码、字模码)？说明这些编码中哪些用二进制编码，哪些不用二进制编码，为什么？

3. 实现下列各数的转换。

(1) $(25.8125)_{10}=(?)_2=(?)_8=(?)_{16}$

(2) $(101101.011)_2=(?)_{10}=(?)_8=(?)_{16}=(?)_{8421}$

(3) $(0101\ 1001\ 0110.0011)_{8421}=(?)_{10}=(?)_2=(?)_{16}$

(4) $(4E.C)_{16}=(?)_{10}=(?)_2$

4. 假定机器数为 8 位(1 位符号，7 位数值)，写出下列各二进制数的原码和补码表示。

+0.1001，−0.1001，+1.0，−1.0，+0.010100，−0.010100，+0，−0

5. 假定机器数为 8 位(1 位符号，7 位数值)，写出下列各二进制数的补码和移码表示。

+1001，−1001，+1，−1，+10100，−10100，+0，−0

6. 已知 $[x]_补$，求 x。

(1) $[x]_补$ =1.1100111　　(2) $[x]_补$ =10000000

(3) $[x]_补$ =0.1010010　　(4) $[x]_补$ =11010011

7. 假定一台32位字长的机器中带符号整数用补码表示，浮点数用IEEE 754标准表示，寄存器R1和R2的内容分别为R1：00 00 10 8BH，R2：80 80 10 8BH。不同指令对寄存器进行不同的操作，因而，不同指令执行时寄存器内容对应的真值不同。假定执行下列运算指令时，操作数为寄存器R1和R2的内容，则R1和R2中操作数的真值分别为多少？

(1) 无符号数加法指令。

(2) 带符号整数乘法指令。

(3) 单精度浮点数减法指令。

8. 假定机器M的字长为32位，用补码表示带符号整数。表2.14第一列给出了在机器M上执行的C语言程序中的关系表达式，请参照已有的表栏内容完成表中后三栏内容的填写。

表2.14　题8用表

关系表达式	运算类型	结果	说　明
0==0U			
−1<0			
−1<0U	无符号整数	0	11…1B ($2^{32}-1$)>00…0B(0)
2 147 483 647>−2 147 483 647−1	有符号整数	1	011…1B ($2^{31}-1$)>100…0B (-2^{31})
2 147 483 647U>−2 147 483 647−1			
2 147 483 647>(int)2 147 483 648U			
−1>−2			
(unsigned)−1>−2			

注：表中第4和第5行的−2 147 483 647−1没有写成−2 147 483 648，因为有些编译器处理一个形如−x的表达式时，可能会先读取表达式x，然后对x取负。当x=2 147 483 648时，因为用32位补码无法表示x，所以，写成−2 147 483 648时可能会发生编译错误。

9. 以下是一个C语言程序，用来计算一个数组a中每个元素的和。当参数len为0时，返回值应该是0，但是在机器上执行时，却发生了存储器访问异常。请问这是什么原因造成的？并说明程序应该如何修改。

```
1 float sum_elements(float a[], unsigned len)
2 {
3     int i;
4     float result=0;
5
6     for (i=0; i <=len-1; i++)
7         result+=a[i];
8
9     return result;
10 }
```

10. 设某浮点数格式如下：

数符	阶码	尾数
1位	5位移码	6位补码

其中,移码的偏置常数为16,补码采用一位符号位,基数为4。

(1) 用这种格式表示下列十进制数:+1.75,+19,−1/8。

(2) 写出该格式浮点数的表示范围,并与12位定点补码整数和定点补码小数表示范围比较。

11. 下列几种情况所能表示的数的范围是什么?

(1) 16位无符号整数

(2) 16位原码定点小数

(3) 16位补码定点小数

(4) 16位补码定点整数

(5) 下述格式的浮点数(基数为2,移码的偏置常数为128):

数符	阶码	尾数
1位	8位移码	7位补码

12. 以IEEE 754单精度浮点数格式表示下列十进制数。

+1.75,+19,−1/8,258

13. 设一个变量的值为4098,要求分别用32位补码整数和IEEE 754单精度浮点格式表示该变量(结果用十六进制表示),并说明哪段二进制序列在两种表示中完全相同,为什么会相同?

14. 设一个变量的值为−2 147 483 647,要求分别用32位补码整数和IEEE 754单精度浮点格式表示该变量(结果用十六进制表示),并说明哪种表示其值完全精确,哪种表示的是近似值。

15. 表2.15给出了有关IEEE 754浮点格式表示中一些重要的非负数的取值,表中已经有最大规格化数的相应内容,要求填入其他浮点数格式的相应内容。

表2.15 题15用表

项目	阶码	尾数	单精度		双精度	
			以2的幂次表示的值	以10的幂次表示的值	以2的幂次表示的值	以10的幂次表示的值
0						
1						
最大规格化数	11111110	1…11	$(2-2^{-23})\times2^{127}$	3.4×10^{38}	$(2-2^{-52})\times2^{1023}$	1.8×10^{308}
最小规格化数						
最大非规格化数						
最小非规格化数						
+∞						
NaN						

16. 已知下列字符编码:A=100 0001,a=110 0001,0=011 0000,求E、e、f、7、G、Z、5的7位ACSII码和第一位前加入奇校验位后的8位编码。

17. 假定在一个程序中定义了变量x、y和i,其中,x和y是float型变量(用IEEE 754单精度浮点数表示),i是16位short型变量(用补码表示)。程序执行到某一时刻,$x=-0.125$、$y=7.5$、$i=100$,它们都被写到了主存(按字节编址),其地址分别是100,108和112。请分别画出在大端机器和小端机器上变量x、y和i在内存中的存放位置。

18. 假定某计算机的总线采用奇校验,每8位数据有一位校验位,若在32位数据线上传输的信息是8F 3C AB 96H,则对应的4个校验位应为什么?若接收方收到的数据信息和校验位分别为87 3C AB 96H和0101B,则说明发生了什么情况?请给出验证过程。

19. 假定一个 16 位数据 M16 M15 M14 M13 M12 M11 M10 M9 M8 M7 M6 M5 M4 M3 M2 M1 为 0101 0001 0100 0110B，要求写出 16 位数据的 SEC 码，并说明 SEC 码如何正确检测数据位 M5 的错误。

20. 假设要传送的数据信息为 100011，若约定的生成多项式为 $G(x)=x^3+1$，则校验码为多少？假定在接收端接收到的数据信息为 100010，说明如何正确检测其错误，写出检测过程。

第 3 章 运算方法和运算部件

计算机完成的功能通过执行程序来实现，任何程序最终都要转换为机器指令。指令中包含的各种算术逻辑运算能直接在硬件上执行，执行这些运算的硬件称为运算部件。

本章首先分析高级语言和机器指令中涉及的各类运算，然后介绍这些运算用到的核心部件——算术逻辑单元(ALU)的组成与工作原理，在此基础上，对这些运算在计算机内部的实现算法和过程进行详细说明，最后介绍实现这些运算的运算部件。因为有些机器也支持十进制加减运算指令，所以，本章最后还将介绍十进制加减运算算法及其运算部件。

3.1 高级语言和机器指令中的运算

计算机硬件的设计目标来源于软件需求。高级语言中用到的各种运算，通过编译成底层的算术运算指令和逻辑运算指令实现，这些底层运算指令能在机器硬件上直接被执行。因此，在介绍运算部件的设计之前，有必要先了解一下高级语言程序和机器指令所涉及到的一些运算。所有高级语言的运算功能大同小异，某一种语言能代表高级语言的总体情况，因此，本教材以 C 语言中的运算为例进行说明。同样，指令中所涉及的运算也可用一个特定的指令系统来说明，本教材主要介绍 MIPS 指令系统中的运算。

*3.1.1 C 程序中涉及的运算

加、减、乘、除等算术运算是高级语言中必须提供的基本运算，可以有无符号数的算术运算、带符号整数的算术运算和浮点数的算术运算。C 语言中除了这些算术运算以外，还有以下几类基本运算：按位运算、逻辑运算、移位运算、位扩展和位截断运算。

1. 按位运算

C 语言中的按位运算有：符号“|”表示按位 OR 运算；符号“&”表示按位 AND 运算；符号“～”表示按位 NOT 运算；符号“^”表示按位 XOR 运算。按位运算的一个重要运用就是实现掩码(masking)操作，通过与给定的一个位模式进行按位与，可以提取所需要的位，然后可以对这些位进行“置 1”、“清 0”、“1 测试”或“0 测试”等。这里位模式被称为“掩码”。例如，表达式“0x0F&0x8C”的运算结果为“00001100”，即“0x0C”。这里通过掩码“0x0F”提取

了一个字节的低4位。

2. 逻辑运算

C语言中的逻辑运算符有：符号“‖”表示OR运算；符号“&&”表示AND运算；符号“!”表示NOT运算，逻辑运算很容易和按位运算混淆，事实上它们的功能完全不同。逻辑运算是非数值计算，其操作数只有两个逻辑值True和False，通常用非0数表示逻辑值True，而全0数表示逻辑值False；而按位运算是一种数值运算，运算时将两个操作数中对应各二进位按照指定的逻辑运算规则逐位进行计算。

3. 移位运算

C语言中提供了一组移位运算。移位操作有逻辑移位和算术移位两种。逻辑移位不考虑符号位，总是把高(低)位移出，低(高)位补0。对于无符号整数的逻辑左移，如果最高位移出的是1，则发生溢出。因为计算机内部的带符号整数都是用补码表示的，所以对于带符号整数的移位操作应采用补码算术移位方式。左移时，高位移出，低位补0，如果移出的高位不同于移位后的符号位，即左移前、后符号位不同，则发生“溢出”；右移时，低位移出，高位补符号。

虽然C语言没有明确规定应该采用逻辑移位还是算术移位。但是，实际上许多机器和编译器都对无符号整数采用逻辑移位方式，而对带符号整数采用算术移位方式。表达式“$x \ll k$”表示对数x左移k位。事实上，对于左移来说，逻辑移位和算术移位的结果都一样，都是丢弃k个最高位，并在低位补k个0。表达式“$x \gg k$”表示对数x右移k位。

每左移一位，相当于数值扩大一倍，所以左移可能会发生溢出，左移k位，相当于数值乘以2^k；每右移一位，相当于数值缩小一半，右移k位，相当于数值除以2^k。

例3.1 已知32位寄存器R1中存放的变量x的机器码为80 00 00 04H，请回答以下问题。

(1) 当x是unsigned int类型时，x的值是多少？$x/2$的值是多少？$x/2$存放在R1中的机器码是什么？$2x$的值是多少？$2x$存放在R1中的机器码是什么？

(2) 当x是signed int类型时，x的值是多少？$x/2$的值是多少？$x/2$存放在R1中的机器码是什么？$2x$的值是多少？$2x$存放在R1中的机器码是什么？

解：(1) 当x是unsigned int类型时，x是无符号数，机器码80 00 00 04H的真值是：

$$+1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100\text{B}=2^{31}+2^{2}$$

对于$x/2$的情况：

一方面，根据x的真值可求得$x/2$的值为$(2^{31}+2^{2})/2=2^{30}+2$；另一方面，$x/2$的机器码可由$x$逻辑右移一位得到，因此，$x/2$存放在R1中的机器码是：

$$0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010=40\ 00\ 00\ 02\text{H}$$

由上述机器码得$x/2$的真值为$2^{30}+2$。因此，根据x的真值求出的$x/2$与由x的机器码逻辑右移得到的$x/2$的机器码求得的$x/2$是一样的。

对于$2x$的情况：

一方面，根据x的真值可求得$2x$的值为$(2^{31}+2^{2})\times 2=2^{32}+2^{3}$；另一方面，$2x$的机器

码由 x 逻辑左移一位得到，因此，$2x$ 存放在 R1 中的机器码是：

0000 0000 0000 0000 0000 0000 0000 1000＝00 00 00 08H

由上述机器码得 $2x$ 的真值为 $2^3=8$。显然 $2^{32}+2^3\neq8$，说明结果溢出。

实际上，对 x 左移时，移出的一位为 1，表明有效数据被丢弃，结果将会溢出，导致根据 x 的真值求出的 $2x$ 与由 x 逻辑左移得到的 $2x$ 的机器码求得的 $2x$ 的值不一样。其原因在于 $2x$ 的值($2^{32}+2^3$)超出了最大可表示数($2^{32}-1$)，无法用 32 位表示。

(2) 当 x 是 signed int 类型时，x 是带符号数，机器码 80 00 00 04H(用二进制补码表示)为：1000 0000 0000 0000 0000 0000 0000 0100。

根据由补码求真值的简便方法"若符号位为 1，则真值的符号为负，其数值部分的各位由补码中相应各位取反后，末尾加 1 所得"，得到 x 的真值为：

$$-0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100\text{B}=-(2^{31}-2^2)$$

对于 $x/2$ 的情况：

一方面，根据 x 的真值可求得 $x/2$ 的值为 $-(2^{31}-2^2)/2=-(2^{30}-2)$；另一方面，$x/2$ 的机器码可由 x 算术右移一位得到，因此，$x/2$ 存放在 R1 中的机器码是：

1100 0000 0000 0000 0000 0000 0000 0010＝C0 00 00 02H

由上述机器码得 $x/2$ 的真值为：

$$-0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\text{B}=-(2^{30}-2)$$

由此可见，由 x 的真值求出的 $x/2$ 与由 x 算术右移得到的 $x/2$ 的机器码求得的 $x/2$ 是一样的。

对于 $2x$ 的情况：

一方面，根据 x 的真值求得 $2x$ 的值为 $-(2^{31}-2^2)\times2=-(2^{32}-2^3)$；另一方面，$2x$ 的机器码可由 x 算术左移一位得到，因此，$2x$ 存放在 R1 中的机器码是：

0000 0000 0000 0000 0000 0000 0000 1000＝00 00 00 08H

由上述机器码得 $2x$ 的真值为 $2^3=8$。显然，$-(2^{32}-2^3)\neq8$，说明结果溢出。

实际上，对 x 左移时，移出的 1 不等于左移后的最高位 0，表明有效数据被丢弃，结果将会溢出，导致根据 x 的真值求出的 $2x$ 与由 x 的机器码算术左移得到的 $2x$ 的机器码求得的 $2x$ 的值不一样。

其原因在于 $2x$ 的真值 $-(2^{32}-2^3)$ 比最小可表示数 -2^{31} 还要小，无法用 32 位表示。

4. 位扩展和位截断运算

C 语言中没有明确的位扩展运算符，但是在进行数据类型转换时，如果遇到一个短数向长数转换，就要进行位扩展运算了。进行位扩展时，扩展后的数值应保持不变。有两种位扩展方式：0 扩展和符号扩展。0 扩展用于无符号数，只要在短的无符号数前面添加足够的 0 即可。符号扩展用于补码表示的带符号整数。通过在短的带符号整数前添加足够多的符号位来扩展。

考虑以下 C 语言程序代码：

```
1 short si=-12345;
```

```
2 unsigned short usi=si;
3 int i=si;
4 unsingned ui=usi ;
```

执行上述程序段，并在32位大端方式机器上输出变量si、usi、i、ui的十进制和十六进制值，可得到各变量的输出结果为：

si＝－12345　　CF C7

usi＝53191　　CF C7

i＝－12345　　FF FF CF C7

ui＝53191　　00 00 CF C7

由此可见，－12345的补码表示和53191的无符号数表示具有相同的16位0/1序列，分别将它们扩展为32位后，得到的32位序列的高位不同。因为前者是符号扩展，高16位补符号1，后者是0扩展，高16位补0。

位截断发生在将长数转换为短数时，例如，对于下列代码：

```
1 int i=53191;
2 short si=(short)i;
3 int j=si;
```

在一台32位大端方式机器上执行上述代码时，第二行要求强行将一个32位带符号数截断为16位带符号整数，53191的32位补码表示为00 00 CF C7H，截断为16位后变成CF C7H，它是－12345的16位补码表示。再将该16位带符号整数扩展为32位时，就变成了FF FF CF C7H，它是－12345的32位补码表示，因此j为－12345。也就是说，原来的i(值为53191)经过截断、再扩展后，其值变成了－12345，不等于原来的值了。

从上述例子可以看出，截断一个数可能会因为溢出而改变它的值。因为长数的表示范围远远大于短数的表示范围，所以当一个长数足够大到短数无法表示的程度，则截断时就会发生溢出。上述例子中的53191大于16位补码能表示的最大数32767，所以就发生了截断错误。这里所说的截断溢出和截断错误只会导致程序出现意外的计算结果，但并不导致任何异常或错误报告。

*3.1.2　MIPS指令中涉及的运算

高级语言中的所有运算都是通过指令系统中的运算指令实现的，一个指令系统中涉及运算的指令有很多，表3.1列出了MIPS指令系统中大部分涉及运算的指令。

从表3.1可以看出，MIPS指令系统涉及的运算有按位逻辑运算、逻辑移位、算术移位、带符号整数的加减乘除、无符号整数加减乘除、带符号整数的符号扩展、无符号数的0扩展、单精度浮点数加减乘除、双精度浮点数加减乘除等。MIPS指令中没有专门的算术左移指令。因为对于左移来说，逻辑移位和算术移位的结果都一样，都是丢弃k个最高位，并在低位补k个0，所以，带符号整数和无符号整数的左移都可用逻辑左移指令实现。

表3.1给出的不是全部MIPS指令，还有一些没有列出。很明显，利用MIPS提供的这些运算指令完全能够实现C语言所需要的各种运算要求。

表 3.1　MIPS 指令系统中涉及运算的部分指令

<table>
<tr><th>指令类型</th><th>指令名称</th><th>汇编形式举例</th><th>含　　义</th><th>所需运算</th></tr>
<tr><td rowspan="7">逻辑运算</td><td>and</td><td>and $1,$2,$3</td><td>$1=$2 & $3</td><td>按位与</td></tr>
<tr><td>or</td><td>or $1,$2,$3</td><td>$1=$2 | $3</td><td>按位或</td></tr>
<tr><td>nor</td><td>nor $1,$2,$3</td><td>$1=~($2 | $3)</td><td>按位或非</td></tr>
<tr><td>and immediate</td><td>andi $1,$2,100</td><td>$1=$2 & 100</td><td>按位与</td></tr>
<tr><td>or immediate</td><td>ori $1,$2,100</td><td>$1=$2 | 100</td><td>按位或</td></tr>
<tr><td>shift left logical</td><td>sll $1,$2,10</td><td>$1=$2 << 10</td><td>逻辑左移</td></tr>
<tr><td>shift right logical</td><td>srl $1,$2,10</td><td>$1=$2 >> 10</td><td>逻辑右移</td></tr>
<tr><td rowspan="11">定点算术运算*</td><td>shift right arithmetic</td><td>sra $1,$2,10</td><td>$1=$2 >> 10</td><td>算术右移</td></tr>
<tr><td>add</td><td>add $1,$2,$3</td><td>$1=$2+$3</td><td>整数加(判溢出)</td></tr>
<tr><td>subtract</td><td>sub $1,$2,$3</td><td>$1=$2−$3</td><td>整数减(判溢出)</td></tr>
<tr><td>add immediate</td><td>addi $1,$2,100</td><td>$1=$2+100</td><td>符号扩展整数加(判溢出)</td></tr>
<tr><td>add unsigned</td><td>addu $1,$2,$3</td><td>$1=$2+$3</td><td>整数加(不判溢出)</td></tr>
<tr><td>subtract unsigned</td><td>subu $1,$2,$3</td><td>$1=$2−$3</td><td>整数减(不判溢出)</td></tr>
<tr><td>add immediate unsigned</td><td>addiu $1,$2,100</td><td>$1=$2+100</td><td>0 扩展整数加(不判溢出)</td></tr>
<tr><td>multiply</td><td>mult $2,$3</td><td>Hi, Lo=$2×$3</td><td>32 位带符号整数乘</td></tr>
<tr><td>multiply unsigned</td><td>multu $2,$3</td><td>Hi, Lo= $2×$3</td><td>32 位无符号整数乘</td></tr>
<tr><td>divide</td><td>div $2,$3</td><td>Lo=$2 ÷ $3
Hi=$2 mod $3</td><td>32 位带符号整数除
Lo=商, Hi=余数</td></tr>
<tr><td>divide unsigned</td><td>divu $2,$3</td><td>Lo=$2 ÷ $3
Hi=$2 mod $3</td><td>32 位无符号整数除
Lo=商, Hi=余数</td></tr>
<tr><td rowspan="6">定点比较、分支</td><td>branch on equal</td><td>beq $1,$2,25</td><td>if($1==$2) goto PC+4+100</td><td>整数减判 0(或其他比较操作),符号扩展,乘 4(左移),整数加</td></tr>
<tr><td>branch on not equal</td><td>bne $1,$2,25</td><td>if($1!=$2) goto PC+4+100</td><td>整数减判 0(或其他比较操作),符号扩展,乘 4(左移),整数加</td></tr>
<tr><td>set on less than</td><td>slt $1,$2,$3</td><td>if($2<$3)$1=1;
else $1=0</td><td>带符号整数比较</td></tr>
<tr><td>set less than immediate</td><td>slti $1,$2,100</td><td>if($2<100)$1=1;
else $1=0</td><td>符号扩展,带符号整数比较</td></tr>
<tr><td>set less than unsigned</td><td>sltu $1,$2,$3</td><td>if($2<$3)$1=1;
else $1=0</td><td>无符号数比较</td></tr>
<tr><td>set less than immediate unsigned</td><td>sltiu $1,$2,100</td><td>if($2<100)$1=1;
else $1=0</td><td>符号扩展,无符号数比较</td></tr>
<tr><td rowspan="5">定点数据传送</td><td>load word</td><td>lw $1,100($2)</td><td>$1=mem[$2+100]</td><td>符号扩展并整数加</td></tr>
<tr><td>store word</td><td>sw $1,100($2)</td><td>mem[$2+100]=$1</td><td>符号扩展并整数加</td></tr>
<tr><td>load half unsigned</td><td>lhu $1,100($2)</td><td>$1=mem[$2+100]</td><td>符号扩展并整数加,0 扩展</td></tr>
<tr><td>store half</td><td>sh $1,100($2)</td><td>mem[$2+100]=$1</td><td>符号扩展并整数加,符号扩展</td></tr>
<tr><td>load byte unsigned</td><td>lbu $1,100($2)</td><td>$1=mem[$2+100]</td><td>符号扩展并整数加,0 扩展</td></tr>
</table>

续表

指令类型	指令名称	汇编形式举例	含　义	所需运算
定点数据传送	store byte	sb $1,100($2)	mem[$2+100]=$1	符号扩展并整数加,符号扩展
	load upper immediate	lui $1,100	$1=100×2^16	逻辑左移16位
浮点算术运算	FP add single	add.s $f2,$f4,$f6	$f2=$f4+$f6	单精度浮点加
	FP subtract single	sub.s $f2,$f4,$f6	$f2=$f4−$f6	单精度浮点减
	FP multiply single	mul.s $f2,$f4,$f6	$f2=$f4×$f6	单精度浮点乘
	FP divide single	div.s $f2,$f4,$f6	$f2=$f4÷$f6	单精度浮点除
	FP add double	add.d $f2,$f4,$f6	$f2=$f4+$f6	双精度浮点加
	FP subtract double	sub.d $f2,$f4,$f6	$f2=$f4−$f6	双精度浮点减
	FP multiply double	mul.d $f2,$f4,$f6	$f2=$f4×$f6	双精度浮点乘
	FP divide double	div.d $f2,$f4,$f6	$f2=$f4÷$f6	双精度浮点除
浮点数据传送	load word corp.1	lwcl $f1,10($2)	$f1=mem[$2+10]	符号扩展并整数加
	load word corp.1	swcl $f1,10($2)	mem[$2+10]=$f1	符号扩展并整数加
浮点比较分支**	branch on FP true	bclt 25	if (cond==1)goto PC+4+100	符号扩展,乘4(左移),整数加
	branch on FP false	bclf 25	if (cond==0) goto PC+4+100	符号扩展,乘4(左移),整数加
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if($f2<$f4) cond=1 else cond=0	单精度浮点减
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if($f2<$f4) cond=1 else cond=0	双精度浮点减

注*：add 和 addu 指令的执行结果相同,只是 add 指令需要检测是否发生补码整数运算溢出,而 addu 指令无须判断溢出。addi 和 addiu 除了在溢出判断上有差别外,立即数的扩展也有差别,addi 中的立即数被看成带符号整数,采用符号扩展,而 addiu 中的立即数被看成无符号数,采用0扩展。减法的情况与加法一样,而乘、除法指令都不判断溢出(通过新增的伪指令来实现溢出判断),但区分无符号数和带符号整数的乘除运算。

注**：MIPS 中有专门的浮点标志 cond,浮点比较和分支指令中用到它,比较指令根据条件设置 cond,分支指令根据 cond 的值,决定是否转移,这些指令生成的条件和判断的条件都默认在 cond 中,所以无须在指令中明显给出。

3.2　基本运算部件

对于逻辑运算指令和定点加减运算指令,实现比较简单。可直接用基本逻辑门电路实现逻辑运算,用定点补码加法器实现无符号数和带符号整数的加减运算。一般情况下,用一个专门的算术逻辑部件(Arithmetic and Logic Unit,ALU)来完成基本逻辑运算和定点数加减运算,各类定点乘除运算和浮点数运算则可利用加法器或 ALU 和移位器来实现,因此基本的运算部件是加法器、ALU 和移位器,ALU 的核心部件是加法器。

ALU 中也可实现左(右)移一位和两位的操作,当然也可用一个移位寄存器实现移位。但这两种方式每次都只能固定移动一位或两位,有时移位指令要求一次移动若干位,对于这种一次左移或右移多位的操作,通常用一个专门的桶形移位器实现。桶形移位器不同于普通移位寄存器,它利用大量多路选择器来实现数据的快速移位,移位操作能够一次完成。在 ALU 外单独设置桶形移位器,还可简化 ALU 的控制逻辑,并实现移位操作和 ALU 操作的

并行性。

3.2.1 串行进位加法器

全加器用来实现两个本位数加上低位进位生成一位本位和及一位向高位的进位。第 i 位的加法运算是第 i 位的加数 X_i、Y_i 和低位 $i-1$ 位的进位 C_{i-1} 三者相加，得到本位和 F_i 和第 i 位的进位输出 C_i。

F_i 和 C_i 的逻辑表达式如下：

$$\left.\begin{aligned} F_i &= X_i \oplus Y_i \oplus C_{i-1} \\ C_i &= X_i C_{i-1} + Y_i C_{i-1} + X_i Y_i \end{aligned}\right\} \tag{3-1}$$

F_i、C_i 被分别称为“全加和”和“全加进位”。图 3.1 和图 3.2 分别是“全加和”和“全加进位”生成电路，从图 3.2 可看出，进位 C_{i-1} 到 C_i 的延迟是两级门。

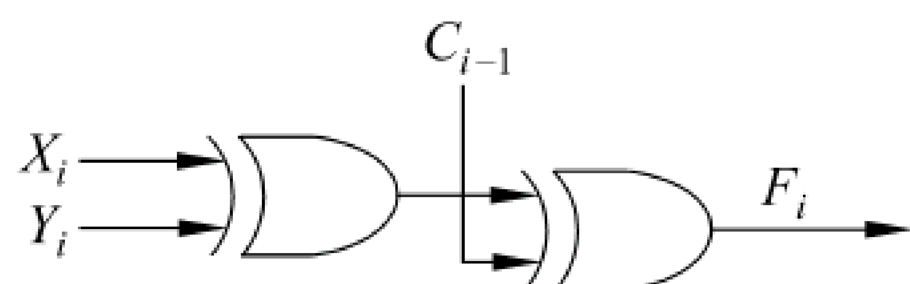


图 3.1 全加和的生成电路

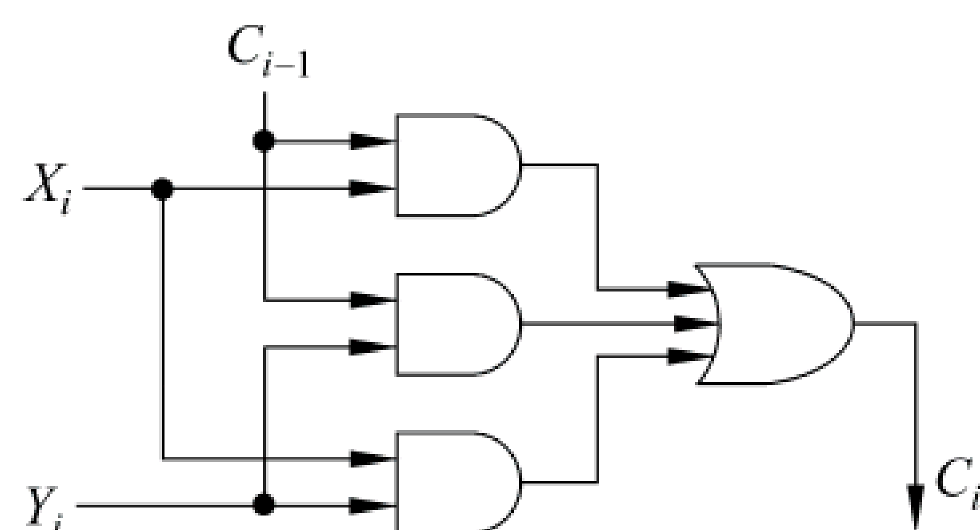


图 3.2 全加进位的生成电路

图 3.3 是全加器的符号表示。将 n 个全加器相连可得 n 位加法器，如图 3.4 所示的加法器实现了两个 n 位二进制数 $X=X_n X_{n-1}\cdots X_1$ 和 $Y=Y_n Y_{n-1}\cdots Y_1$ 逐位相加的功能，得到的二进制和为 $F=F_n F_{n-1}\cdots F_1$，进位输出为 C_n。例如，当 $X=11\cdots11$，$Y=00\cdots01$ 时，最后的输出为 $F=00\cdots00$ 且 $C_n=1$。由于只有有限位数，高位自动丢失，所以实际上是在模 2^n 运算系统下的加法运算，可以实现 n 位无符号数的加法和 n 位补码加法。

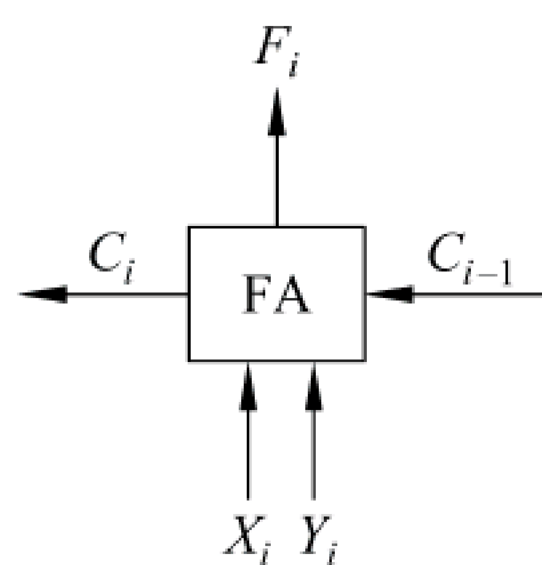


图 3.3 全加器符号

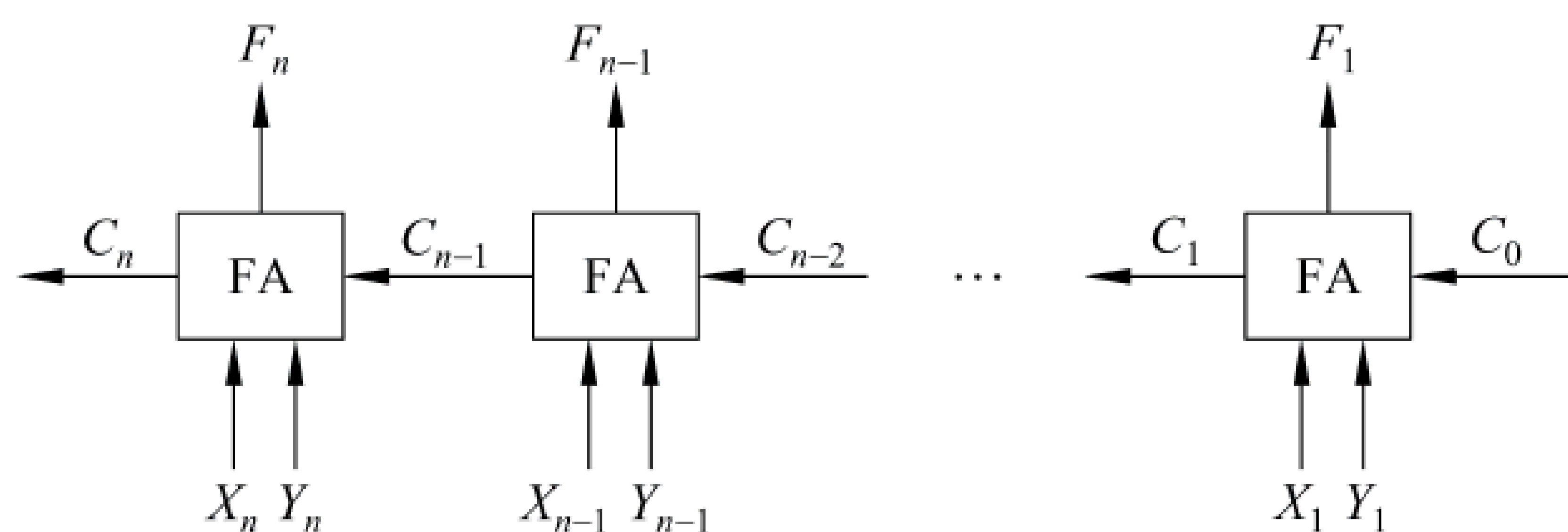


图 3.4 n 位串行进位加法器

对于图 3.4 所示的 n 位加法器，X 与 Y 逐位相加，只有当进位信号顺序地从最低位传到最高位，才能稳定地形成最后的输出。这种加法器的位间进位是串行传送的。任意一位加法运算，都必须等到低位加法做完送来进位时才能正确进行，因此称为串行进位方式。我们知道，一块小石头扔进平静的水中，泛起的波纹会向外一圈一圈逐步扩散，串行进位加法器中最低进位 C_0 就像一块小石头，它把进位逐步从低位扩展到最高位，所以，这种串行进位的加法器被称为行波进位加法器(Carry Ripple Adder，CRA)。

这种结构所用元件较少，但进位传递时间较长。从图3.2可以看出，全加器内从进位输入到进位输出，共经过两级门延迟，所以，n位串行加法器从C_0到C_n的延迟时间为$2n$级门延迟。假定一个异或门为三级门延迟，则最后一位和数F_n的延迟时间为$2n+1$级门延迟。加法运算时间随两个加数位数n的增加而增加。当n较大时，串行进位的加法器速度将显著变慢。

前面说过，几乎所有的算术运算都要用到ALU或加法器，ALU的核心还是加法器，因此要提高运算速度，加法器的速度非常关键。由于串行进位加法器速度慢的主要原因是进位按串行方式传递，高位进位依赖低位进位。为了提高加法器的速度，必须尽量避免进位之间的依赖关系。下面介绍的进位选择加法器(Carry Select Adder)和并行进位加法器都在某种程度上加快了运算速度。

*3.2.2 进位选择加法器

进位选择加法器通过提前进行高位部分的运算来使高、低两部分并行执行以加快运算速度，因为低位部分向高位部分的进位还没有得到，所以在进行高位部分的加运算时，同时用两个加法器实现，一个加法器的低位进位为0，另一个为1，等到低位部分的结果出来后，根据低位部分向高位部分的进位来选择使用哪个加法器的结果。

图3.5和图3.6分别给出了8位串行进位加法器和8位进位选择加法器示意图。进位选择加法器通过用两个高位部分加法器代替一个高位加法器，实现了高、低两部分的并行执行，可使运算时间减半。同理，还可继续把4位加法器分解成高、低各两位，使运算时间再减半。

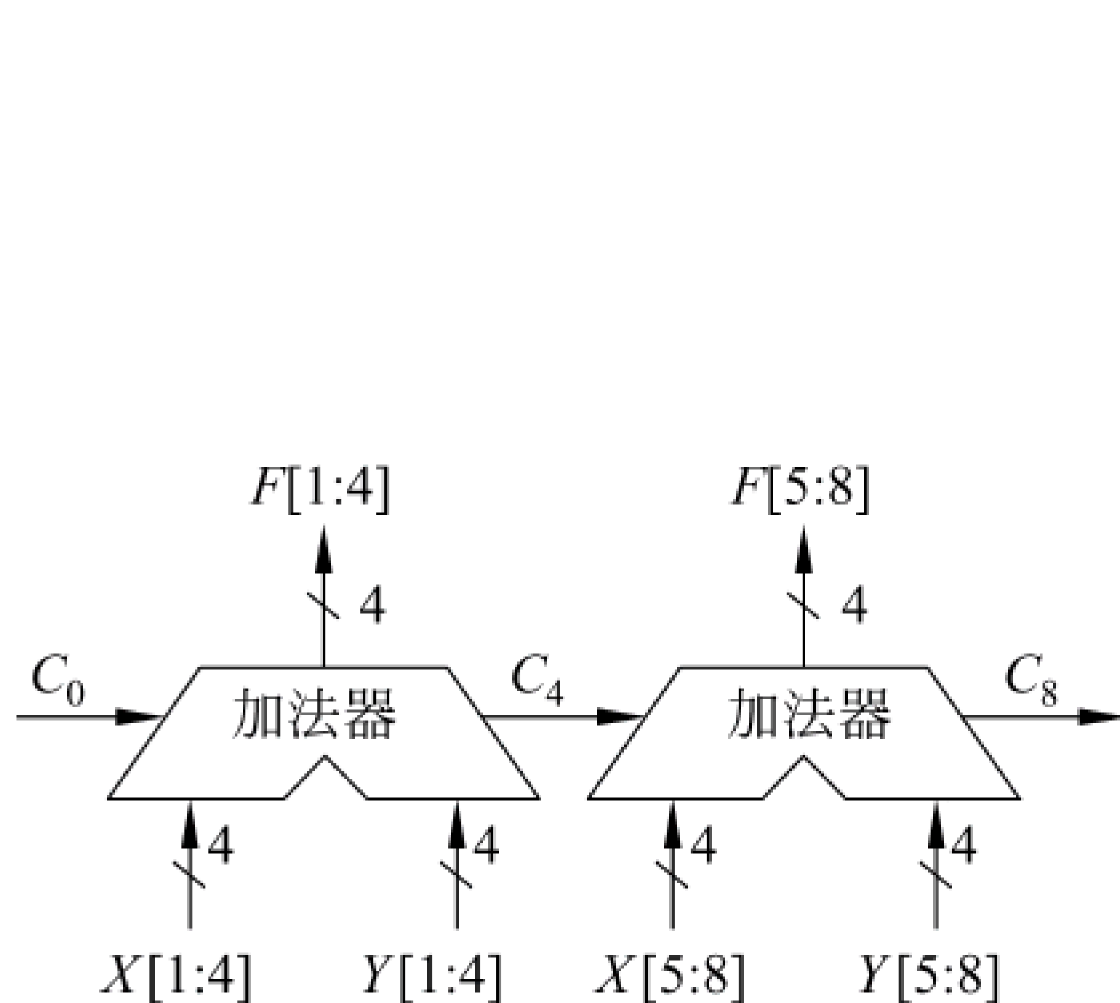


图3.5 串行进位加法器

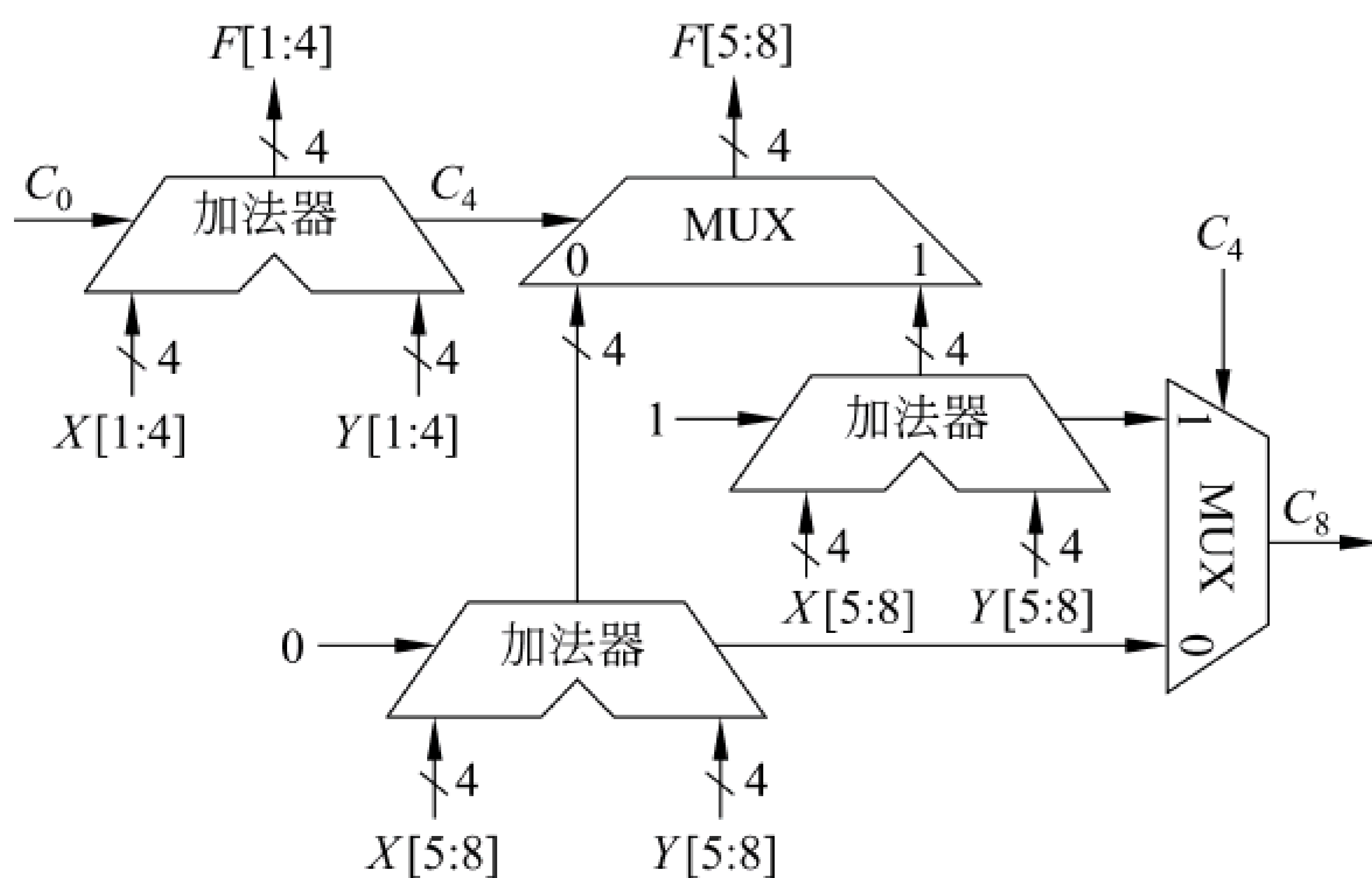


图3.6 进位选择加法器

进位选择加法器通过重叠设置加法器来加快速度，因而代价较大。下面介绍的超前进位加法器采用并行进位方式，通过让进位之间减少依赖，使各进位独立且并行产生，以加快运算速度。

3.2.3 并行进位加法器

由全加器公式(3-1)可知，对于一个4位加法器，其进位C_1、C_2、C_3和C_4的产生条件

如下：

$$
\begin{aligned}
C_1 &= X_1Y_1 + (X_1 + Y_1)C_0 \\
C_2 &= X_2Y_2 + (X_2 + Y_2)C_1 \\
&= X_2Y_2 + (X_2 + Y_2)X_1Y_1 + (X_2 + Y_2)(X_1 + Y_1)C_0 \\
C_3 &= X_3Y_3 + (X_3 + Y_3)C_2 \\
&= X_3Y_3 + (X_3 + Y_3)[X_2Y_2 + (X_2 + Y_2)X_1Y_1 + (X_2 + Y_2)(X_1 + Y_1)C_0] \\
&= X_3Y_3 + (X_3 + Y_3)X_2Y_2 + (X_3 + Y_3)(X_2 + Y_2)X_1Y_1 \\
&\quad + (X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1)C_0 \\
C_4 &= X_4Y_4 + (X_4 + Y_4)C_3 \\
&= X_4Y_4 + (X_4 + Y_4)[X_3Y_3 + (X_3 + Y_3)X_2Y_2 + (X_3 + Y_3)(X_2 + Y_2)X_1Y_1 \\
&\quad + (X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1)C_0] \\
&= X_4Y_4 + (X_4 + Y_4)X_3Y_3 + (X_4 + Y_4)(X_3 + Y_3)X_2Y_2 \\
&\quad + (X_4 + Y_4)(X_3 + Y_3)(X_2 + Y_2)X_1Y_1 \\
&\quad + (X_4 + Y_4)(X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1)C_0
\end{aligned}
$$

从以上公式来看，每个进位表达式中都含有(X_i+Y_i)和X_iY_i，所以，定义两个辅助函数如下：

$$
\left.\begin{aligned}
P_i &= X_i + Y_i \\
G_i &= X_iY_i
\end{aligned}\right\} \tag{3-2}
$$

P_i称为进位传递函数，其含义是：当X_i、Y_i中有一个为1时，若低位有进位输入，则一定被传递到高位。这个进位可看作是低位进位越过本位直接向高位传递的。G_i称为进位生成函数，其含义是：当X_i、Y_i均为1时，不管有无低位进位输入，本位一定向高位产生进位输出。

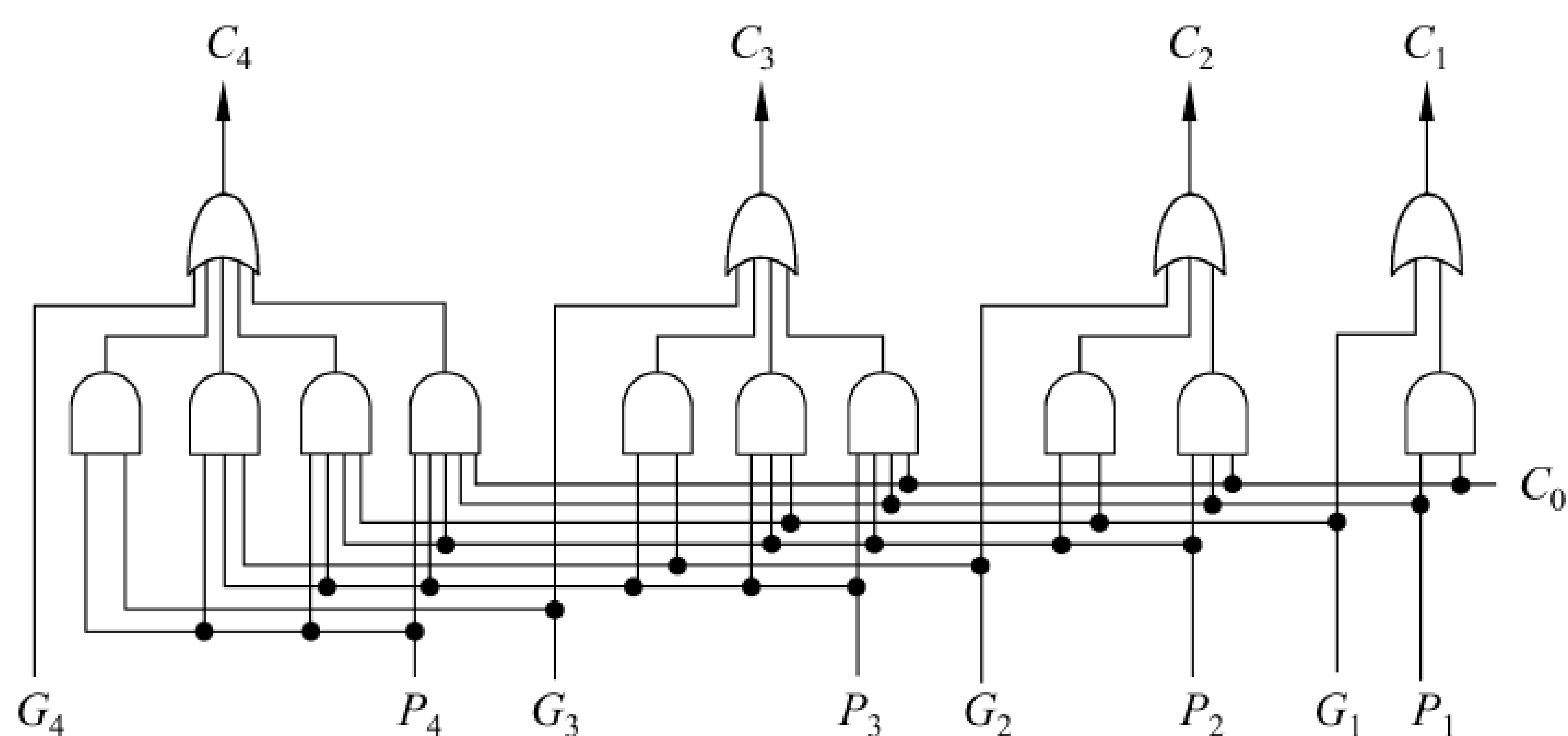
将P_i、G_i代入前面$C_1 \sim C_4$式中，可得：

$$
\left.\begin{aligned}
C_1 &= G_1 + P_1C_0 \\
C_2 &= G_2 + P_2G_1 + P_2P_1C_0 \\
C_3 &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0 \\
C_4 &= G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1 + P_4P_3P_2P_1C_0
\end{aligned}\right\} \tag{3-3}
$$

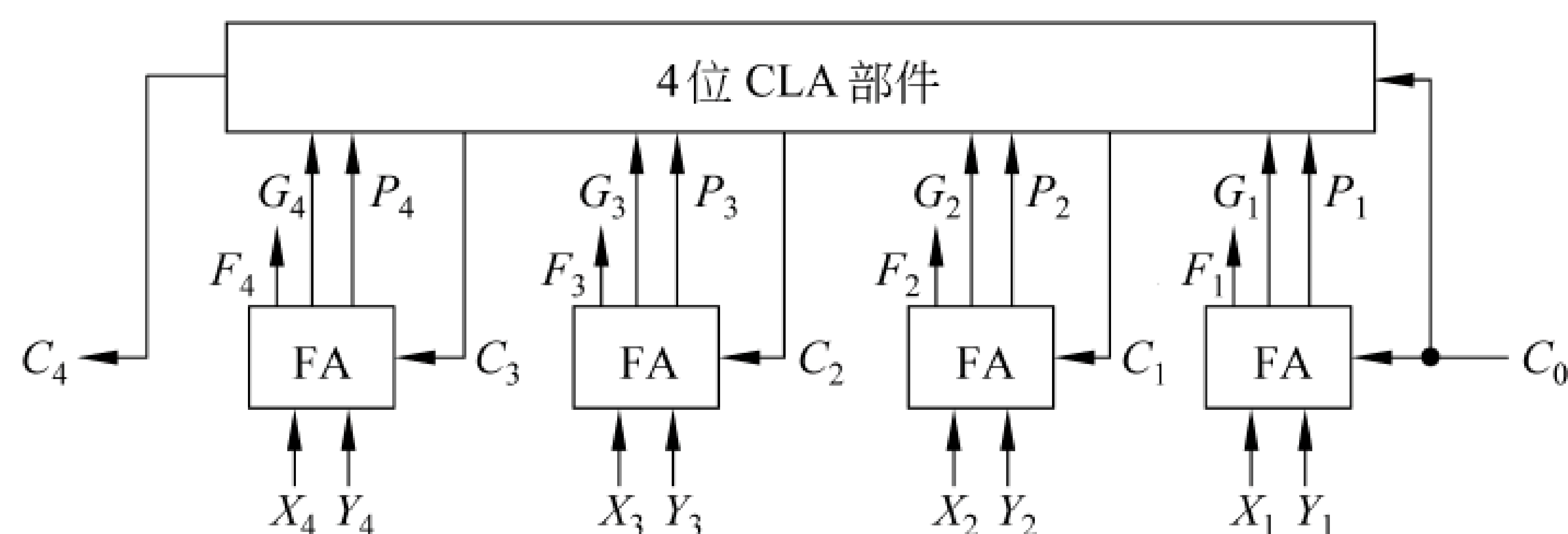
从上述表达式可以看出，C_i仅与X_i、Y_i和C_0有关，与相互间的进位无关。只要$X_1 \sim X_4$、$Y_1 \sim Y_4$和C_0同时到达，就可几乎同时形成$C_1 \sim C_4$，并同时生成各位的和。

实现上述逻辑表达式(3-3)的电路称为先行进位(也称超前进位)部件(Carry Lookahead Unit)，也称CLA部件。通过这种进位方式实现的加法器称为全先行进位加法器。因为各个进位是并行产生的，所以是一种并行进位加法器。图3.7为4位CLA部件和4位全先行进位加法器示意图。

由图3.7可看出，从X_i、Y_i到产生P_i、G_i需要1级门延迟，从P_i、G_i、C_0到产生所有进位$C_1 \sim C_4$需要2级门延迟，产生全部和需要1+2+3=6级门延迟(假定一个异或门等于3级门延迟)。所以4位全先行进位加法器的关键路径长度为6级门延迟。



(a) 4位 CLA 部件



(b) 4位全先行进位加法器

图 3.7　4 位 CLA 部件和 4 位全先行进位加法器

从公式(3.3)可知,更多位数的 CLA 部件只会增加逻辑门的输入端个数,而不会增加门的级数,因此,如果用全先行进位方式构建更多位数的加法器,从理论上讲,应该还是 6 级门延迟。但是由于 CLA 部件中连线数量和输入端个数的增多,使得实现电路中需要具有大驱动信号和大扇入门。因而,当位数较多时,全先行进位实现方式不太现实。例如,对于一个 32 位全先行进位加法器,其生成 C_{32} 的与门和或门有 30 多个输入端。因此更多位数的加法器可通过 4 位 CLA 部件或 4 位全先行进位加法器来实现。

用 4 位全先行进位加法器串接起来可构成 $4n$ 位加法器,这种并行加法器称为单级先行进位加法器。如图 3.8 所示是一个 16 位单级先行进位加法器,分成 4 组,每组对应一个 4 位全先行进位加法器。

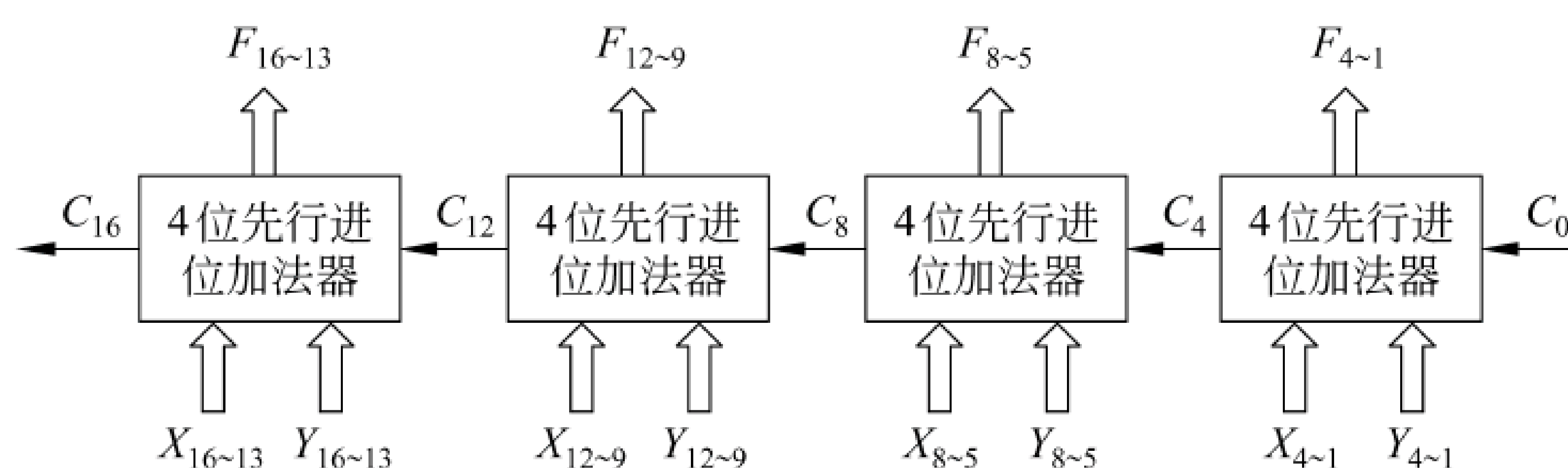


图 3.8　16 位单级先行进位加法器

单级先行进位加法采用“组间串行、组内并行”的进位方式。从图 3.8 可看出,其关键路径为($X_{4\sim1}$、$Y_{4\sim1}$、C_0)→(C_4)→(C_8)→(C_{12})→($F_{16\sim13}$),关键路径长度为 3+2+2+5=12 级门延迟。对于 $4n$ 位单级先行进位加法器,其延迟为 $2n+4$,而 $4n$ 位串行进位的行波加法器

延迟为 $2\times4n+1=8n+1$，因此速度大约提高了 4 倍。

为了进一步提高加法器的运算速度，可以进一步采用组内和组间都并行的进位方式。将式(3-3)中进位 C_4 的逻辑方程改写为：

$$C_4 = G_{m1} + P_{m1}C_0 \tag{3-4}$$

C_4 表示 4 位加法器的进位输出，P_{m1}、G_{m1} 分别表示 4 位加法器的进位传递输出和进位生成输出，分别为：

$$P_{m1} = P_4P_3P_2P_1$$

$$G_{m1} = G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1$$

将式(3-4)应用于 4 个 4 位先行进位加法器，则有：

$$\left.\begin{aligned}C_4 &= G_{m1} + P_{m1}C_0\\C_8 &= G_{m2} + P_{m2}C_4 = G_{m2} + P_{m2}G_{m1} + P_{m2}P_{m1}C_0\\C_{12} &= G_{m3} + P_{m3}C_8 = G_{m3} + P_{m3}G_{m2} + P_{m3}P_{m2}G_{m1} + P_{m3}P_{m2}P_{m1}C_0\\C_{16} &= G_{m4} + P_{m4}C_{12} = G_{m4} + P_{m4}G_{m3} + P_{m4}P_{m3}G_{m2} + P_{m4}P_{m3}P_{m2}G_{m1} + P_{m4}P_{m3}P_{m2}P_{m1}C_0\end{aligned}\right\} \tag{3-5}$$

比较式(3-3)和式(3-5)，可以看出这两组进位逻辑表达式是类似的。不过，式(3-3)表示的是组内进位，式(3-5)表示的是组间进位。实现逻辑方程组(3-5)的电路被称为成组先行进位(Block Carry Lookahead，BCLA)部件，也称 BCLA 部件。这种组内和组间都采用并行进位的加法器被称为两级先行进位加法器。用类似的方式可以构建多级先行进位加法器。如图 3.9 所示，是一个由 4 个 4 位先行进位加法器与 1 个 BCLA 部件构成的 16 位两级先行进位加法器。

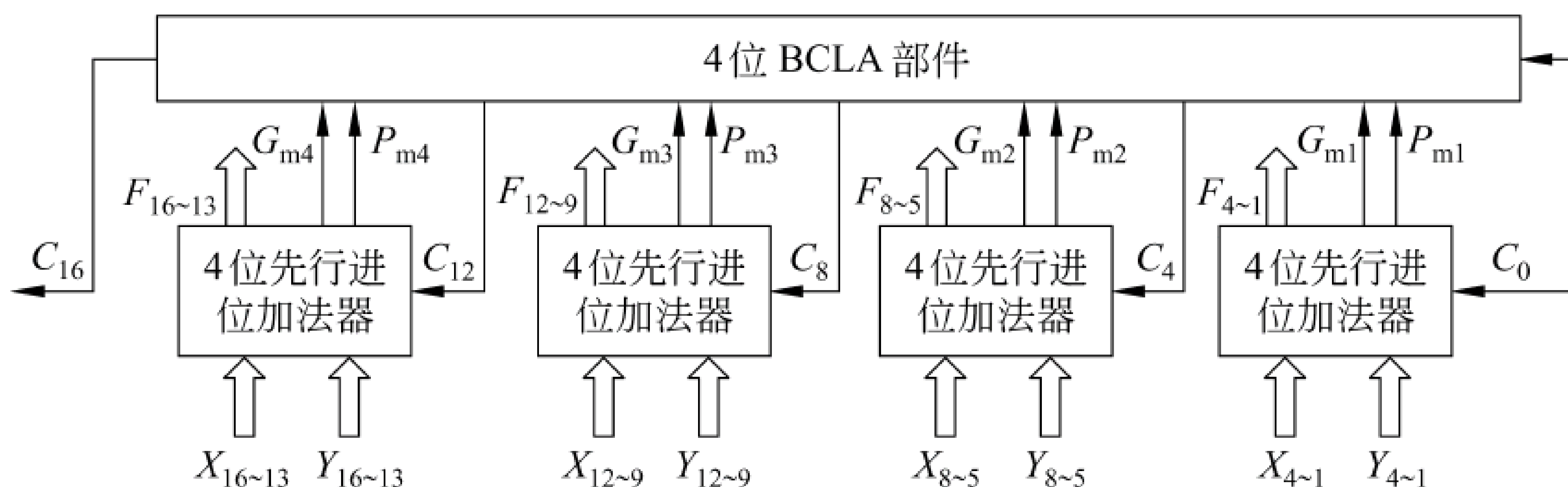


图 3.9　16 位两级先行进位加法器

从图 3.9 可看出，16 位两级先行进位加法器中的关键路径为$(X_i、Y_i)\to(P_{mi}、G_{mi}、C_0)\to(C_{4i})\to(F_i)$，关键路径长度为 $3+2+3=8$ 级门延迟，最终进位 C_{16} 的延迟为 $3+2=5$ 级门。

因为两级先行进位加法器组内和组间都采用先行进位方式，其延迟和加法器的位数没有关系，不会随着位数的增加而延长时间。所以，计算机内部大多采用两级或多级先行进位加法器。

3.2.4　算术逻辑部件

ALU 是一种能进行多种算术运算与逻辑运算的组合逻辑电路，它的基本逻辑结构是加

法器，通常用图 3.10 所示的符号来表示。其中 A 和 B 是两个 N 位操作数输入端，CarryIn 是进位输入端，ALUop 是操作控制端，用来决定 ALU 所执行的处理功能。例如，ALUop 选择 Add 运算，ALU 就执行加法运算，输出的结果就是 A 加 B 之和。ALUop 的位数决定了操作的种类，例如，当位数为 3 时，ALU 最多只有 8 种操作。Result 是运算结果输出端，此外，还有相应的运算结果标志信息：零标志（Zero）、溢出标志（Overflow）和进位标志（CarryOut）。

图 3.11 给出了能够完成 3 种运算“与”、“或”和“加法”的一位 ALU 结构图。其中，一位加法用一个全加器（Full Adder）实现，在 ALUop 的控制下，由一个多路选择器（Mux）选择输出 3 种操作结果之一。这里有 3 种操作，所以 ALUop 至少要有两位。在一位 ALU 基础上，可以利用串行进位或单级、多级先行进位等方式构造多位 ALU。正如前面所说，先行进位方式比串行进位方式速度快，因此，ALU 多采用先行进位方式。图 3.12 是采用先行进位方式构建的 4 位单级先行进位 ALU 示意图。

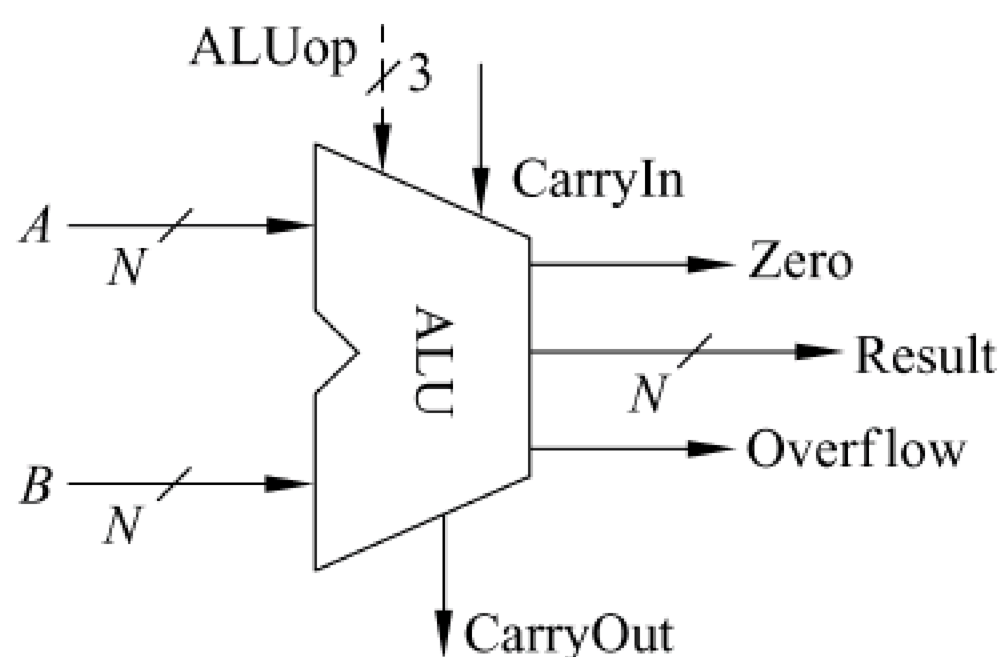


图 3.10　ALU 符号

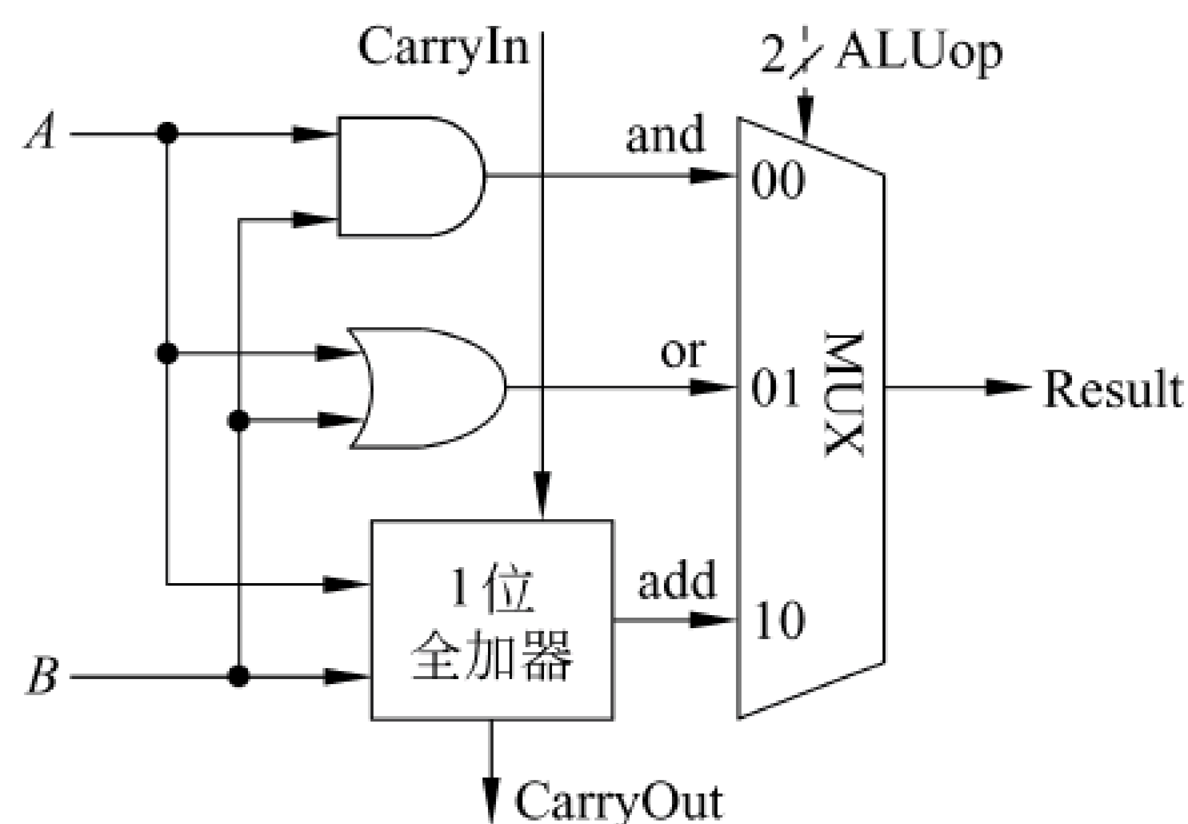


图 3.11　一位 ALU 结构

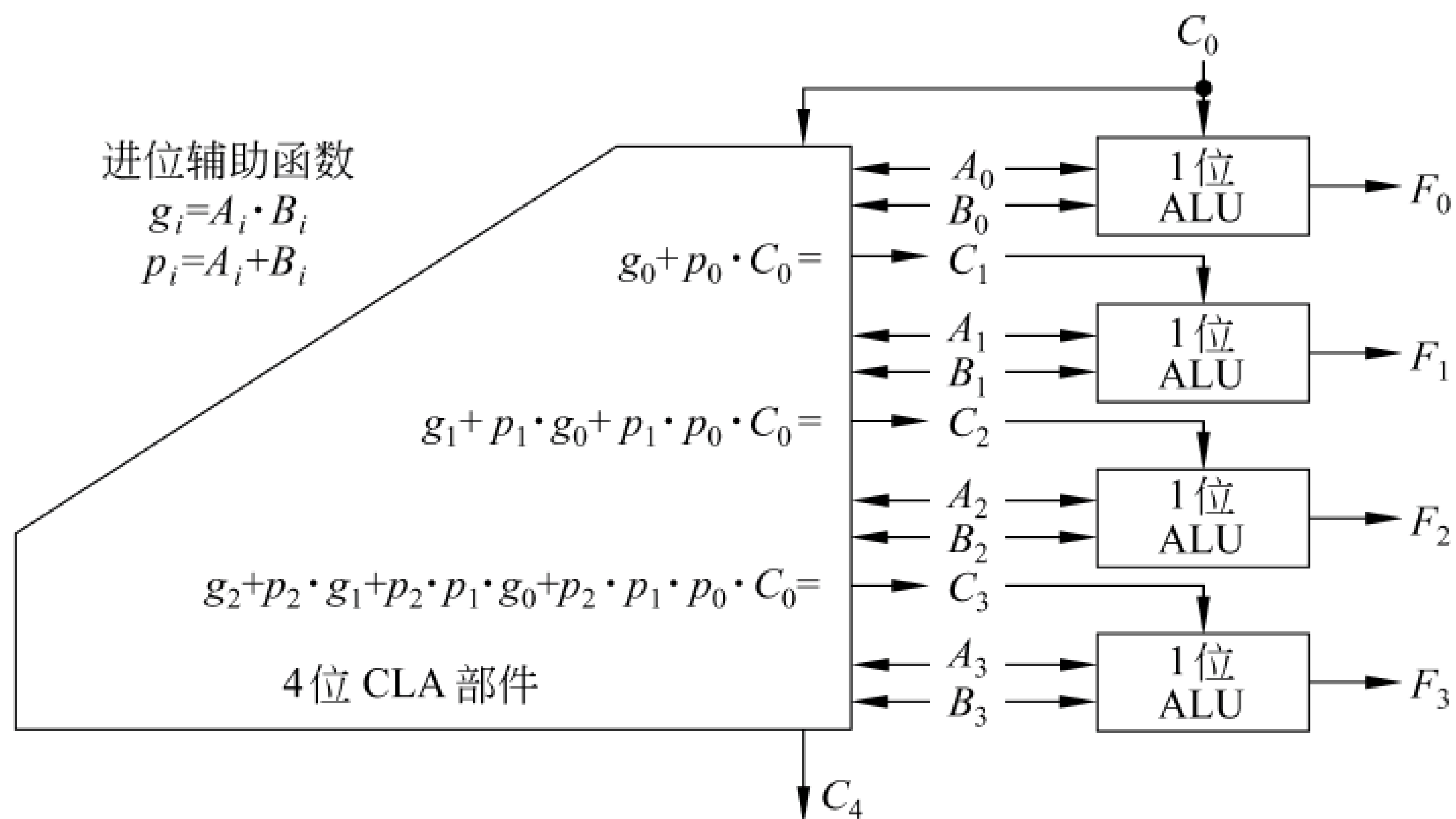


图 3.12　4 位单级先行进位 ALU

SN74181 是早期的一个典型中规模商售 4 位 ALU 组件。它能对两个 4 位二进制代码 $A_3A_2A_1A_0$ 和 $B_3B_2B_1B_0$ 进行 16 种算术运算（当 M 为低电位时）和 16 种逻辑运算（当 M 为高电位时），产生结果 $F_3F_2F_1F_0$。这两类各 16 种运算操作由 $S_3S_2S_1S_0$ 四位功能选择端控制。C_n 是 ALU 的最低位进位输入，低电平信号有效，即 $C_n=L$ 表示有进位输入，C_{n+4} 是 ALU 进位输出信号。

SN74181 有正逻辑和负逻辑两种芯片。表 3.2 是 SN74181 在正逻辑下的功能表。表中“加”指加法，“+”指逻辑或。从表 3.2 可看出，当 $S_3S_2S_1S_0$ 为 1001 时进行算术加运算，此时，C_n 为高电平，即最低位进位为 0。当 $S_3S_2S_1S_0$ 为 0110 时进行算术减运算，此时，C_n 为低电平，即最低位进位为 1。除了加、减运算外，还可实现取反、与、或、非、或非、异或、恒 1、恒 0、直送等各种运算。

表 3.2　SN74181 ALU 功能表

S_3	S_2	S_1	S_0	M=H 逻辑运算	M=L 算术运算	
					$\bar{C}_n=1(C_n=H)$	$\bar{C}_n=0(C_n=L)$
L	L	L	L	$\overline{A}$	A	$A+1$
L	L	L	H	$\overline{A+B}$	$A+B$	$(A+B)$加 1
L	L	H	L	$\overline{A}\cdot B$	$A+\overline{B}$	$(A+\overline{B})$加 1
L	L	H	H	“0”	减 1	“0”
L	H	L	L	$\overline{A\cdot B}$	A 加$(A\cdot\overline{B})$	A 加$(A\cdot\overline{B})$ 加 1
L	H	L	H	$\overline{B}$	$(A\cdot\overline{B})$加$(A+B)$	$(A\cdot\overline{B})$加$(A+B)$加 1
L	H	H	L	$A\oplus B$	A 减 B 减 1	A 减 B
L	H	H	H	$A\cdot\overline{B}$	$(A\cdot\overline{B})$减 1	$A\cdot\overline{B}$
H	L	L	L	$\overline{A}+B$	A 加$(A\cdot B)$	A 加$(A\cdot B)$ 加 1
H	L	L	H	$\overline{A\oplus B}$	A 加 B	A 加 B 加 1
H	L	H	L	B	$(A\cdot B)$加$(A+\overline{B})$	$(A\cdot B)$加$(A+\overline{B})$加 1
H	L	H	H	$A\cdot B$	$(A\cdot B)$减 1	$A\cdot B$
H	H	L	L	“1”	A 加 A	A 加 A 加 1
H	H	L	H	$A+\overline{B}$	A 加$(A+B)$	A 加$(A+B)$加 1
H	H	H	L	$A+B$	A 加$(A+\overline{B})$	A 加$(A+\overline{B})$ 加 1
H	H	H	H	A	A 减 1	A

将多片 SN74181 组合，可以构成更多位数的 ALU。在构建更多位数的多级先行进位 ALU 时，要用到组进位生成部件。图 3.13 和图 3.14 分别是 4 位 ALU 部件 SN74181 和 4 位 BCLA 部件 SN74182 的外部特性图。

如图 3.15 所示，用 4 个 SN74181 和一个 SN74182 可构成一个 16 位两级先行进位 ALU。每个 SN74181 能提供相应的 G、P 信号给 SN74182，以实现芯片间进位的并行生成，从而实现芯片之间的并行运算。

类似地利用 16 片 SN74181 和 5 片 SN74182 芯片，采用三级先行进位方式，可以很容易地组成 64 位快速 ALU。当然，对于通用计算机中的处理器，因为 ALU 是处理器芯片中的

内部电路，不可能用像SN74181和SN74182这样的芯片来连接生成ALU，但是ALU的设计思想是一样的。

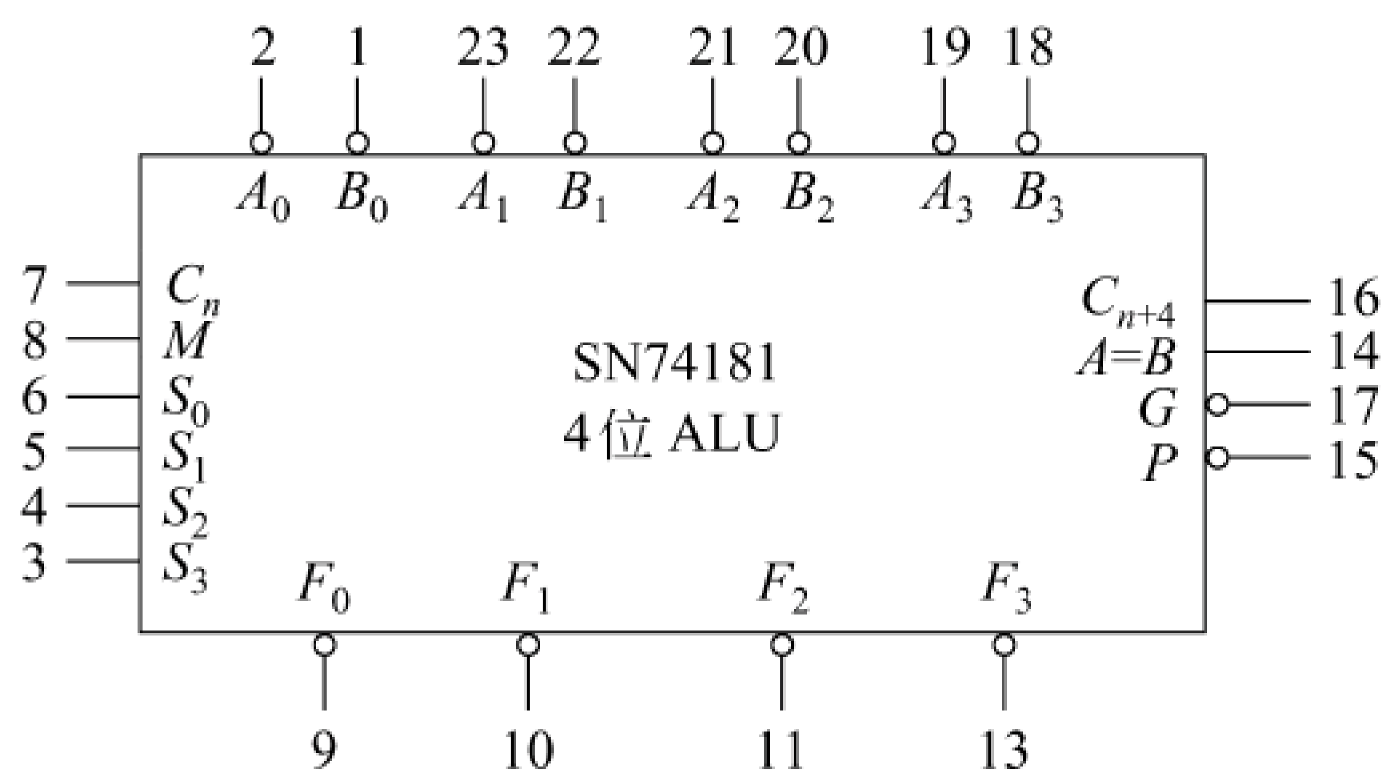


图3.13　SN74181外部特性图

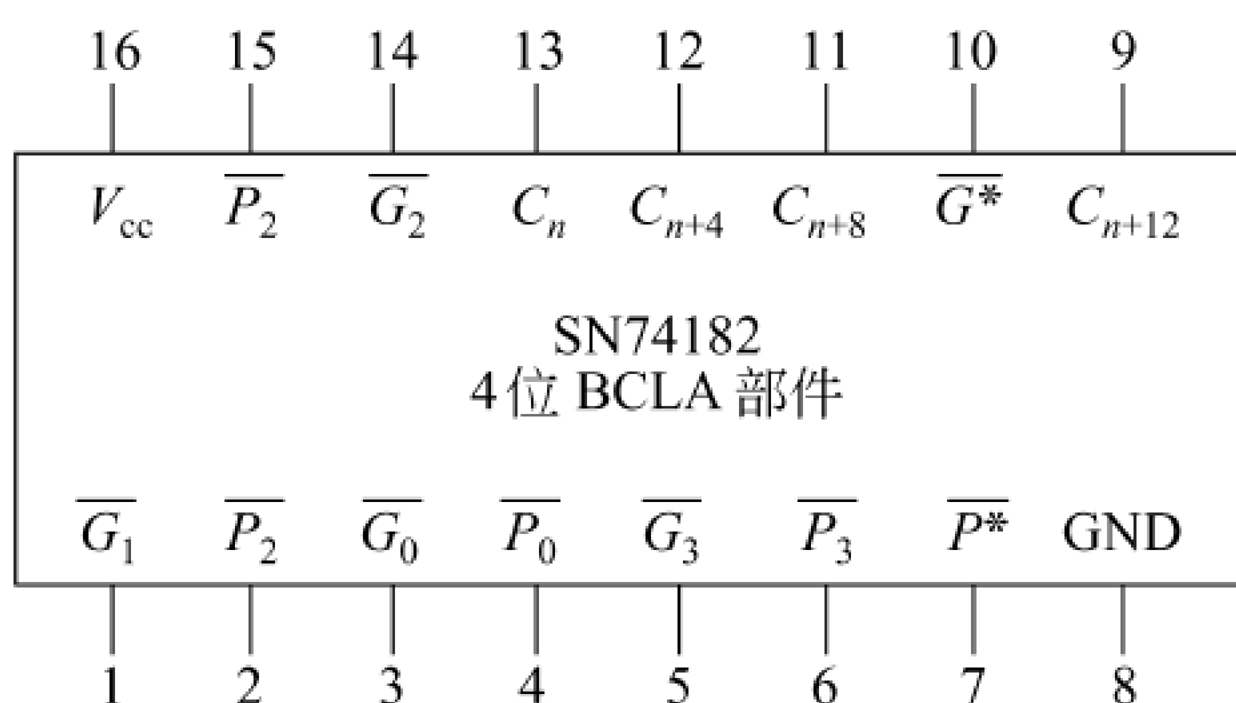


图3.14　SN74182外部特性图

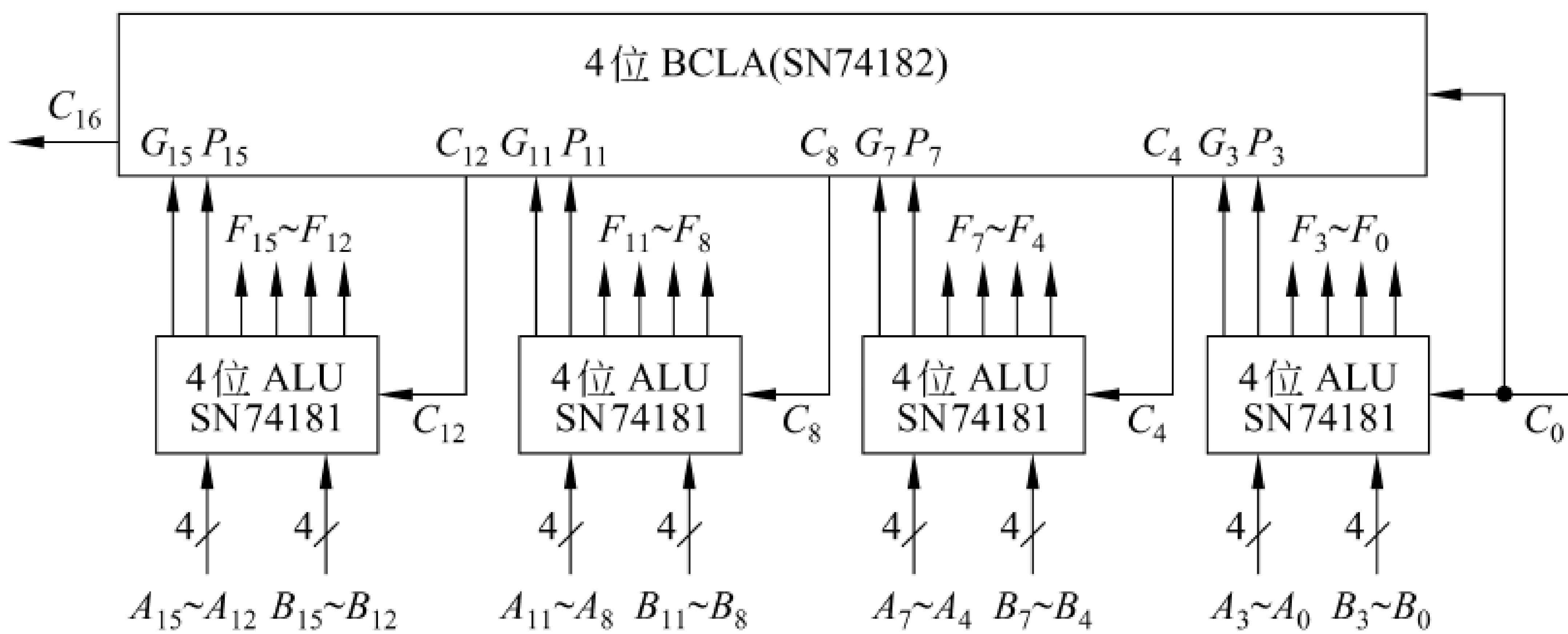


图3.15　16位两级先行进位ALU

3.3　定点数运算

从前面第3.1节介绍的有关高级语言和机器指令涉及到的运算来看，定点运算主要包括：无符号数的按位逻辑运算，无符号数的逻辑移位运算，无符号数的位扩展运算和截断运算、无符号数的加、减、乘、除运算，带符号整数的算术移位运算，带符号整数的扩展运算和截断运算，带符号整数的加、减、乘、除运算等。

无符号数的按位逻辑运算可用逻辑门电路实现，无符号数的逻辑移位运算可用专门的移位器或斜送结果等多种方式来实现，带符号数的移位运算、无符号数和带符号整数的位扩

展运算和截断运算也可用简单电路较容易地实现。

因此，对于无符号数和带符号整数的运算，本节主要内容是加、减、乘、除运算以及这些运算所涉及到的运算部件。计算机内部带符号数基本都是用补码表示的，所以带符号整数的运算主要介绍补码运算。

浮点数由一个定点小数和一个定点整数表示，大多数机器采用 IEEE 754 标准来表示浮点数，IEEE 754 标准用定点原码小数表示尾数，用移码表示阶，因而浮点数运算涉及到原码定点小数的加、减、乘、除运算和移码的加、减运算。因此，本节同时也介绍原码定点小数的加、减、乘、除运算和移码的加减运算。

3.3.1 补码加减运算

若两个补码表示的 n 位定点整数 $[X]_补 = X_{n-1}X_{n-2}\cdots X_0$，$[Y]_补 = Y_{n-1}Y_{n-2}\cdots Y_0$，则 $[X+Y]_补$ 和 $[X-Y]_补$ 的运算表达式如下：

$$\left.\begin{aligned}[X+Y]_补 &= [X]_补 + [Y]_补 \quad (\text{mod } 2^n)\\ [X-Y]_补 &= [X]_补 + [-Y]_补 \quad (\text{mod } 2^n)\end{aligned}\right\} \tag{3-6}$$

运算公式(3-6)的正确性可以从补码的编码规则得到证明。从式(3-6)中可看出，在补码表示方式下，无论 X、Y 是正数还是负数，加、减运算统一采用加法来处理，而且 $[X]_补$ 和 $[Y]_补$ 的符号位(最高有效位 MSB)可以和数值位一起参与运算，加、减运算结果的符号位也在求和运算中直接得出，这样，可以直接用前面第 3.2 节介绍的加法器实现“$[X]_补 + [Y]_补 (\text{mod } 2^n)$”和“$[X]_补 + [-Y]_补 (\text{mod } 2^n)$”。最终运算结果的高位丢弃，保留低 n 位，相当于对和数取模 2^n。因此，实现减法的主要工作在于求 $[-Y]_补$。

根据第 2 章介绍的补码运算的特点，可知：求一个数的负数的补码可以由其补码“各位取反、末位加 1”得到。也即已知一个数的补码表示为 Y，则这个数负数的补码为 $-Y=\overline{Y}+1$，因此，只要在原加法器的 Y 输入端，加 n 个反向器实现各位取反的功能，然后加一个 2 选 1 多路选择器，用一个控制端 Sub 来控制选择将原码 Y 输入到加法器还是将 $\overline{Y}$ 输入到加法器，并将控制端 Sub 同时作为低位进位送到加法器，如图 3.16 所示。该电路可实现补码加减运算。当控制端 Sub 为 1 时，做减法，实现 $X+\overline{Y}+1=X-Y$；当控制端 Sub 为 0 时，做加法，实现 $X+Y$。

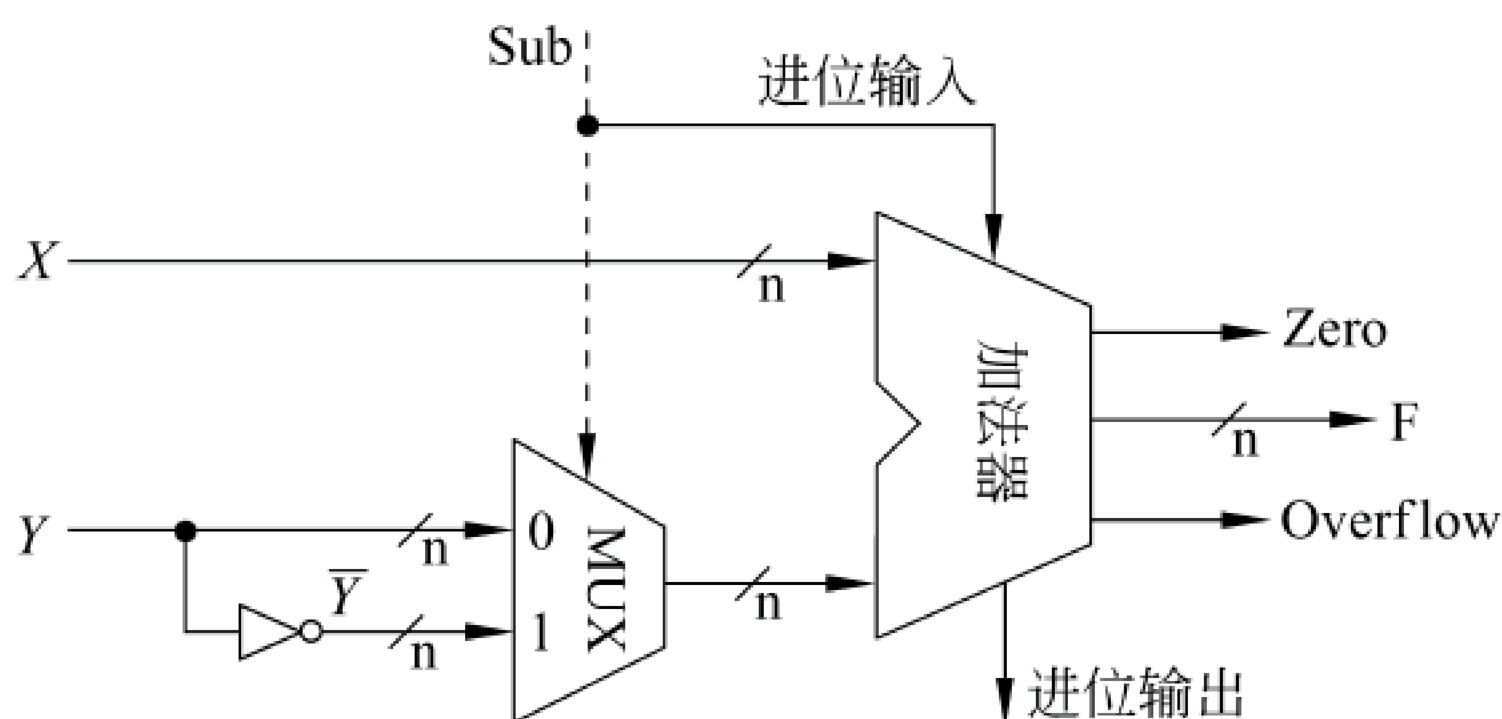


图 3.16　补码加减运算部件

从前面第 3.1 节介绍的 MIPS 指令中的运算可以看出，分支指令(条件转移指令)需要对两个带符号数做减法，然后判断结果是否为 0，以决定是否转移。因此，补码加减运算部件中要有判 0 电路。图 3.17 是判“0”电路示意图。

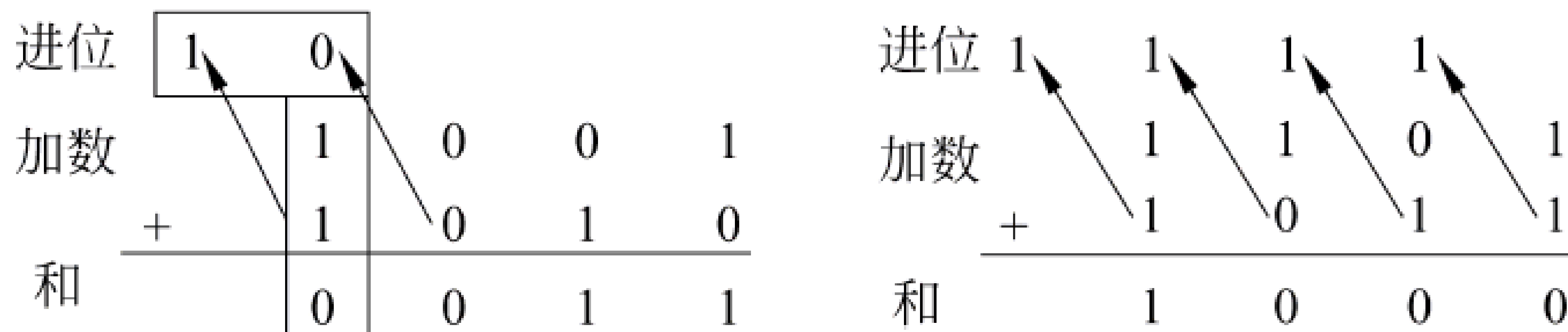
所有指令系统都和 MIPS 的一样，对于带符号数的运算都要进行“溢出”判断，因此，在运算器中应有溢出判别线路和溢出标志位。对于 n 位补码整数，它可表示的数值范围为 $-2^{n-1}\sim2^{n-1}-1$。当运算结果超出该范围时，则结果溢出。补码溢出判断方法有多种，先看两个例子。

例 3.2 用 4 位补码计算“$-7-6$”和“$-3-5$”的值。

解：$[-7]_补=1001$ $[-6]_补=1010$ $[-3]_补=1101$ $[-5]_补=1011$

$[-7-6]_补=[-7]_补+[-6]_补=1001+1010=0011(+3)$

$[-3-5]_补=[-3]_补+[-5]_补=1101+1011=1000(-8)$。



因为 4 位补码的可表示范围为 $-8\sim+7$，而 $-7-6=-13<-8$，所以，结果 0011(+3) 一定发生了溢出，是一个错误的值。

考察“$-7-6$”的例子后发现以下两种现象：

(1) 最高位和次高位的进位不同。

(2) 和的符号位和加数的符号位不同。

对于例子“$-3-5$”，结果 1000(-8)没有超出范围，因而没有发生溢出，是一个正确的值。此时，最高位的进位和次高位的进位都是 1，没有发生第(1)种现象，而且，和的符号和加数的符号都是 1，也没有发生第(2)种现象。

通常根据上述两种现象是否发生来判断有无溢出。因此，有以下两种溢出判断逻辑表达式。

① 若符号位产生的进位 C_n 与最高数值位向符号位的进位 C_{n-1} 不同，则产生溢出，即：

$$\text{Overflow}=C_{n-1}\oplus C_n$$

② 若两个加数的符号位 X_{n-1} 和 Y_{n-1} 相同，且与和的符号位 F_{n-1} 不同，则产生溢出，即：

$$\text{Overflow}=X_{n-1}\,Y_{n-1}\,\overline{F}_{n-1}+\overline{X}_{n-1}\,\overline{Y}_{n-1}\,F_{n-1}$$

根据上述溢出判断逻辑表达式，可以很容易实现溢出判断电路。图 3.18 是上述第①种方法对应的溢出判断电路的示意图。

对于采用变形补码(双符号位补码，模 4 补码)的机器，可以有其他的溢出判断方法。在采用双符号位时，正数的双符号位是 00，负数的双符号位是 11。

两个正数相加时，若不溢出，则数值位不应向符号位产生进位，两个正数的双符号位运算为 00+00=00，结果为正，是正确的和的双符号位；若溢出，则数值位肯定向符号位产生进位，此时，两个正数双符号位的运算为 00+00+1=01，和的双符号位是 01，前面的0 是真正的符号，后面的 1 是溢出到符号位的数值，因此，运算的实际结果为正数，结果为正溢出。

两个负数相加时，若不溢出，则数值位应向符号位产生进位，两个负数的双符号位运算为(11+11+1)mod 4=11，结果为负，正好是正确的和的双符号位；若溢出，则数值位未向

符号位产生进位，此时，两个负数的双符号位的运算为(11+11)mod 4=10，因此，和的双符号位为10，前面的1是真正的符号，后面的0是溢出到符号位的数值部分，因此，运算的实际结果为负数，结果为负溢出。

由此可得变形补码加减运算的溢出判断条件为：若结果的两个符号位 F_n 和 F_{n-1} 不一致，则产生溢出。即：

$$\text{Overflow}=F_{n-1}\oplus F_n$$

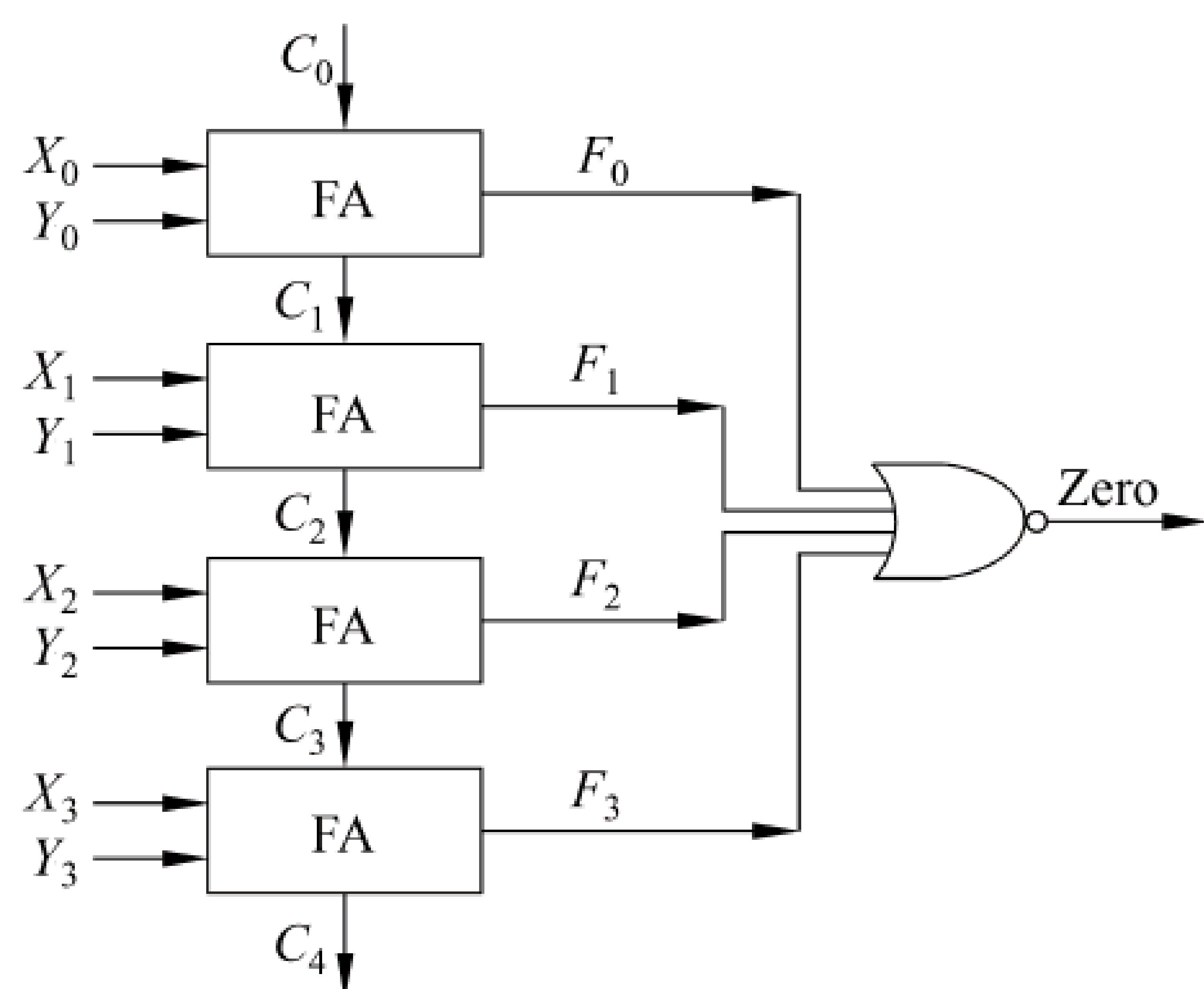


图 3.17 判“0”电路

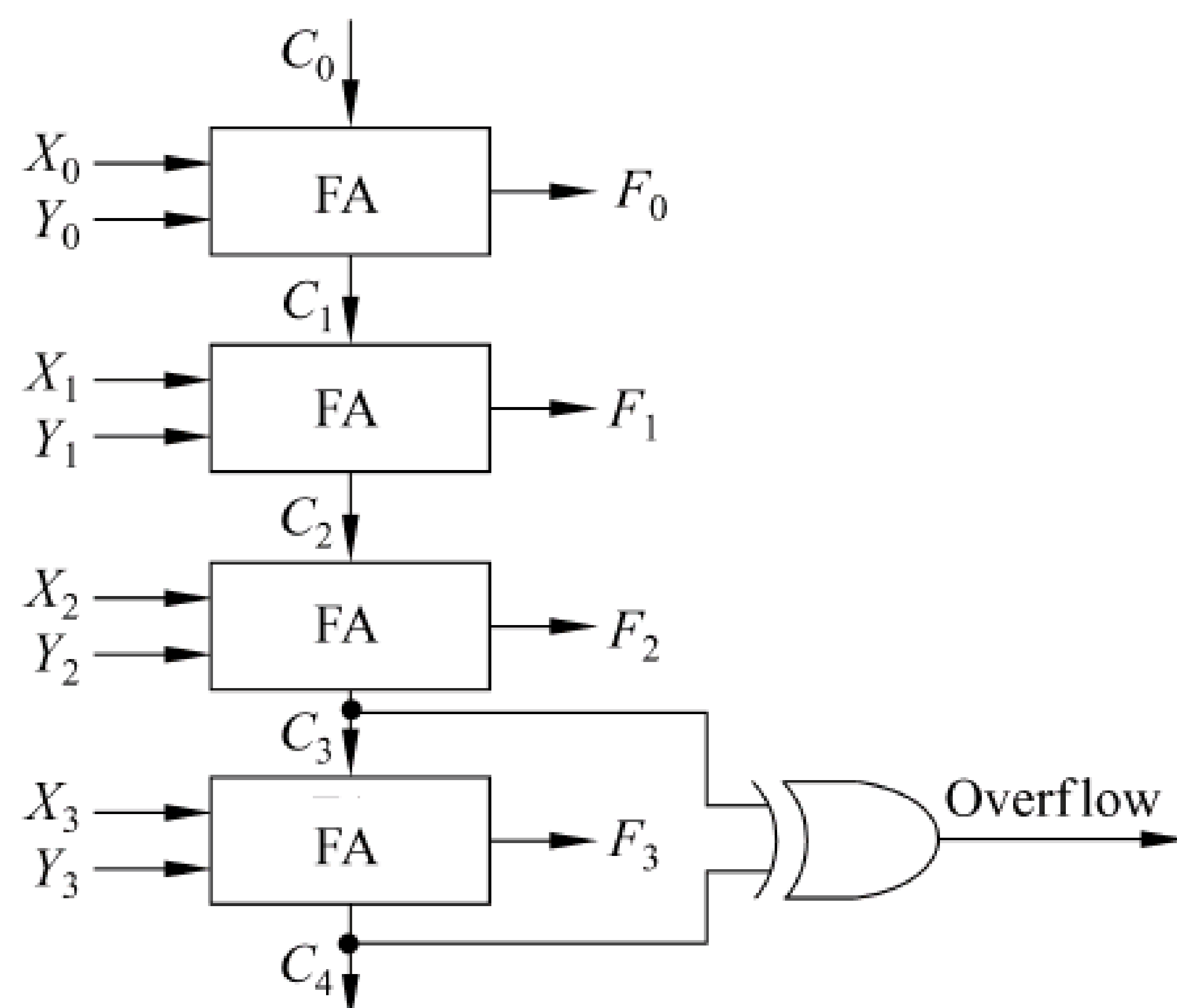


图 3.18 溢出判断电路

对于运算结果，除了最终的和数，以及相应的“0”标志、“溢出”标志外，许多机器还提供进位标志、符号标志等。这些标志信息在运算电路中产生后，被记录到专门的寄存器中，以便在分支指令中被用来作为检测条件。存放这些标志的寄存器通常称为(程序)状态(字)寄存器或标志寄存器。每个标志信号对应标志寄存器中的一个标志位。例如，Intel x86 处理器中有一个标志寄存器 Flag，其中包含与运算有关的标志如下：

CF(Carry Flag) 进位标志：反映运算执行后是否在最高位产生进位或借位。主要用在多字节加减运算中。移位和逻辑运算也可使其产生 CF。若产生进位或借位，则 CF=1，否则，CF=0。

AF(Auxiliary Carry Flag)辅助进位标志：反映运算后是否在低4位产生进位或借位。主要用于 BCD 码加减运算结果的调整。有关调整方法见后面3.7节。若产生进位或借位，则 AF=1，否则，AF=0。

PF(Parity Flag)奇偶标志：反映运算结果低8位的奇偶性。可用于检查数据的奇偶性。若含偶数个1，则 PF=1，否则，PF=0。

ZF(Zero Flag)零标志：反映运算结果是否为0。若结果为0，则 ZF=1，否则，ZF=0。

SF(Sign Flag)符号标志：反映运算结果符号是否为1(负数)。若是负数，则 SF=1，否则，SF=0。

OF(Overflow Flag)溢出标志：反映运算结果是否溢出。若运算结果溢出，则 OF=1，否则，OF=0。

*3.3.2 原码加减运算

计算机中的浮点数多采用 IEEE 754 标准，其尾数用原码表示。所以有必要了解

原码定点小数的运算。在原码加减运算中，符号位和数值位是分开来计算的。符号位在运算过程中起判断和控制作用，并且对结果的符号位产生影响。加减运算在数值位上进行。

原码加减运算规则如下：

(1) 比较两个操作数的符号，对加法实行“同号求和，异号求差”，对减法实行“异号求和，同号求差”。

(2) 求和时，数值位相加，若最高位产生进位，则结果溢出。和的符号位取被加数(或被减数)的符号。

(3) 求差时，被加数(或被减数)的数值位加上加数(或减数)的数值位的补码，并按以下规则产生结果。

① 最高数值位产生进位，表明加法结果为正，所得数值位正确。

② 最高数值位没有产生进位，表明加法结果为负，得到的是数值位的补码形式，因此，需要对结果求补，还原为绝对值形式的数值位。

③ 差的符号位：在上述①的情况下，符号位取被加数(被减数)的符号；在上述②的情况下，符号位为被加数(被减数)的符号取反。

例 3.3 已知 $[X]_{原}=1.0011$，$[Y]_{原}=1.1010$，计算$[X+Y]_{原}$。

解：根据原码加减运算规则可知两数同号，用加法求和，和的符号同被加数的符号。

和的数值位为：$0011+1010=1101$，和的符号位为 1，因此，$[X+Y]_{原}=1.1101$。

例 3.4 已知 $[X]_{原}=1.0011$，$[Y]_{原}=1.1010$，计算$[X-Y]_{原}$。

解：根据原码加减运算规则可知两数同号，用减法求差。

差的数值位为 $0011+(1010)_{补}=0011+0110=1001$，最高数值位没有产生进位，表明加法结果为负，需对 1001 求补，还原为绝对值形式的数值位为$(1001)_{补}=0111$。

差的符号位为$[X]_{原}$的符号位取反，即为 0，所以 $[X-Y]_{原}=0.0111$。

上述运算过程在浮点数运算部件中的尾数加减法器中实现。有关浮点数运算部件详见第 3.5 节。

*3.3.3 移码加减运算

计算机中的浮点数多采用 IEEE 754 标准，其阶码用移码表示。在进行浮点数加减运算中，需要比较两个浮点数阶码的大小，在进行浮点数乘除运算中，需要求阶码的和与差，因此浮点数运算涉及到移码定点加减运算。

假设 E 为阶，所取移码位数为 n，根据如下移码和补码的定义：

$$[E]_{移}=2^{n-1}+E \quad (-2^{n-1}\leqslant E<2^{n-1})$$

$$[E]_{补}=\begin{cases}E & (0\leqslant E<2^{n-1})\\ 2^{n}+E & (-2^{n-1}\leqslant E<0)\end{cases} \quad (\mathrm{mod}\ 2^{n})$$

可以推导出移码的加减运算规则为：

$$[E1]_{移}+[E2]_{移}=2^{n-1}+E1+2^{n-1}+E2=2^{n}+E1+E2=[E1+E2]_{补} \quad (\mathrm{mod}\ 2^{n})$$

$$\begin{aligned}[E1]_{移}-[E2]_{移}&=[E1]_{移}+[-[E2]_{移}]_{补}=2^{n-1}+E1+2^{n}-[E2]_{移}\\&=2^{n-1}+E1+2^{n}-2^{n-1}-E2=2^{n}+E1-E2=[E1-E2]_{补} \quad (\mathrm{mod}\ 2^{n})\end{aligned}$$

由上述规则可知：移码的和、差等于和、差的补码。

根据第2章2.1.4节中介绍的移码和补码仅符号位不同的关系，可得出移码加减运算的步骤如下。

（1）对于加法，直接将$[E1]_{移}$和$[E2]_{移}$进行模2^n相加，然后对结果的符号取反。

（2）对于减法，先将减数$[E2]_{移}$求补（各位取反，末位加1），然后再与被减数$[E1]_{移}$进行模2^n相加，最后对结果的符号取反。

（3）溢出判断规则：进行模2^n相加时，如果两个加数的符号相同，并且与和数的符号也相同，则发生溢出。

例3.5 用4位移码计算"$-7+(-6)$"和"$-3+6$"的值。

解：$[-7]_{移}=0001$ $[-6]_{移}=0010$ $[-3]_{移}=0101$ $[6]_{移}=1110$。

$[-7]_{移}+[-6]_{移}=0001+0010=0011$（两个加数与结果的符号都为0，溢出）。

从移码加法运算过程来看，相加结果0011的符号位取反后为1011，其值为+3，两个负数相加的结果为正数，说明结果发生了溢出，因此，与溢出判断规则得出的结果相同。用十进制运算验证如下：$-7+(-6)=-13$，结果-13小于4位移码可表示的最小数-8，说明结果应该是溢出的。

$[-3]_{移}+[6]_{移}=0101+1110=0011$，符号取反后为1011，其真值为+3。

例3.6 用4位移码计算"$-7-(-6)$"和"$-3-5$"的值。

解：$[-7]_{移}=0001$ $[-6]_{移}=0010$ $[-3]_{移}=0101$ $[5]_{移}=1101$。

$[-7]_{移}-[-6]_{移}=0001+1110=1111$，符号取反后为0111，其真值为$-1$。

$[-3]_{移}-[5]_{移}=0101+0011=1000$，符号取反后为0000，其真值为$-8$。

上述移码加减运算主要用于浮点数乘除运算中的阶码相加减。

3.3.4 原码乘法运算

原码作为浮点数尾数的表示形式，需要计算机能实现定点原码小数的乘法运算。根据每次部分积是一位相乘得到，还是两位相乘得到，可以有原码一位乘法和原码两位乘法，根据原码两位乘法的原理推广，可以有原码多位乘法。

1. 原码一位乘法

用原码实现乘法运算时，符号位与数值位分开计算，因此，原码乘法运算分为两步。

（1）确定乘积的符号位。由两个乘数的符号异或得到。

（2）计算乘积的数值位。乘积的数值部分为两个乘数的数值部分之积。

原码乘法算法描述如下：已知$[X]_{原}=x_0.x_1\cdots x_n$，$[Y]_{原}=y_0.y_1\cdots y_n$，则

$[X\times Y]_{原}=z_0.z_1\cdots z_{2n}$，其中$z_0=x_0\oplus y_0$，$z_1\cdots z_{2n}=(0.x_1\cdots x_n)\times(0.y_1\cdots y_n)$

可以不管小数点，事实上在机器内部也没有小数点，只是约定了一个小数点的位置，小数点约定在最左边就是定点小数乘法，约定在右边就是定点整数乘法。因此，两个定点小数的数值部分之积可以看成是两个无符号数的乘积。

下面是一个手算乘法的例子，以此可以推导出两个无符号数相乘的计算过程。

```
   0.1 0 1 1        被乘数 X=0.x1x2x3x4=0.1 0 1 1
  ×0.1 1 0 1          乘数 Y=0.y1y2y3y4=0.1 1 0 1
  ----------
       1 0 1 1 ······ X×y4×2^-4
     0 0 0 0 ········ X×y3×2^-3
   1 0 1 1 ·········· X×y2×2^-2
 1 0 1 1 ············ X×y1×2^-1
 ------------
 0.1 0 0 0 1 1 1 1
```

由此可知，$X\times Y=\sum_{i=1}^{4}(X\times y_i\times 2^{-i})=0.10001111$

从上述手算乘法过程可以看出，两个无符号数相乘具有以下几个特点。

(1) 用乘数 Y 的每一位依次去乘以被乘数得 $X\times y_i$，$i=4、3、2、1$。若 $y_i=0$，则得 0；若 $y_i=1$，则得 X。

(2) 把(1)中求得的各项结果 $X\times y_i$ 在空间上向左错位排列，即逐次左移，可以表示为 $X\times y_i\times 2^{-i}$。

(3) 对(2)中求得的结果求和，就是两个无符号数的乘积。

计算机中两个无符号数相乘，类似手算乘法。但为了提高效率，做了相应改进。主要的改进措施有以下几个方面。

(1) 每将乘数 Y 的一位乘以被乘数得 $X\times y_i$ 后，就将该结果与前面所得的结果累加，得到 P_i，称之为部分积(Partial Product，PP)。因为没有等到全部计算后一次求和，所以减少了保存每次相乘结果 $X\times y_i$ 的开销。

(2) 在每次求得 $X\times y_i$ 后，不是将它左移与前次部分积 P_i 相加，而是将部分积 P_i 右移一位与 $X\times y_i$ 相加。

(3) 对乘数中为“1”的位执行加法和右移运算，对为“0”的位只执行右移运算，而不需执行加法运算。

因为每次进行加法运算时，只需要将 $X\times y_i$ 与部分积中的高 n 位进行相加，低 n 位不会改变，因此，只需用 n 位加法器就可实现两个 n 位数的相乘。

上述思想可以写成数学推导过程如下：

$$
\begin{aligned}
X\times Y&=X\times(0.y_1y_2\cdots y_n)\\
&=X\times y_1\times 2^{-1}+X\times y_2\times 2^{-2}+X\times y_3\times 2^{-3}+\cdots+X\times y_n\times 2^{-n}\\
&=\underbrace{2^{-1}(2^{-1}(2^{-1}\cdots 2^{-1}(2^{-1}}_{n\text{个}2^{-1}}(0+X\times y_n)+X\times y_{n-1})+\cdots+X\times y_2)+X\times y_1)
\end{aligned}
$$

上述推导过程具有明显的递归性质，其递推公式为：

$$P_{i+1}=2^{-1}(P_i+X\times y_{n-i})\quad(i=0,1,2,3,\cdots,n-1)\tag{3-7}$$

设 $P_0=0$，无符号数乘法过程可以归结为循环地计算下列算式的过程。

$$
\begin{aligned}
P_1&=2^{-1}(P_0+X\times y_n)\\
P_2&=2^{-1}(P_1+X\times y_{n-1})\\
&\vdots\\
P_n&=2^{-1}(P_{n-1}+X\times y_1)
\end{aligned}
$$

上述推导过程中的 P_i 称为部分积，每一步迭代过程如下。

(1) 取乘数的最低位 y_{n-i} 判断。

(2) 若 y_{n-i} 为 1，则将第(1)步迭代部分积 P_i 与 X 相加；若 y_{n-i} 为 0，则什么也不做。

(3) 右移一位，产生本次部分积 P_{i+1}。

部分积 P_i 和 X 进行无符号数相加，可能会产生进位，因而需要有一个专门的进位位 C。整个迭代过程从乘数最低位 y_n 和 $P_0=0$ 开始，经过 n 次"判断—加法—右移"循环，直到求出 P_n 为止。P_n 就是最终的乘积。假定每次循环在一个时钟周期内完成，则 n 位乘法需要用 n 个时钟周期来完成。图 3.19 是实现两个 32 位无符号数乘法的逻辑电路图。

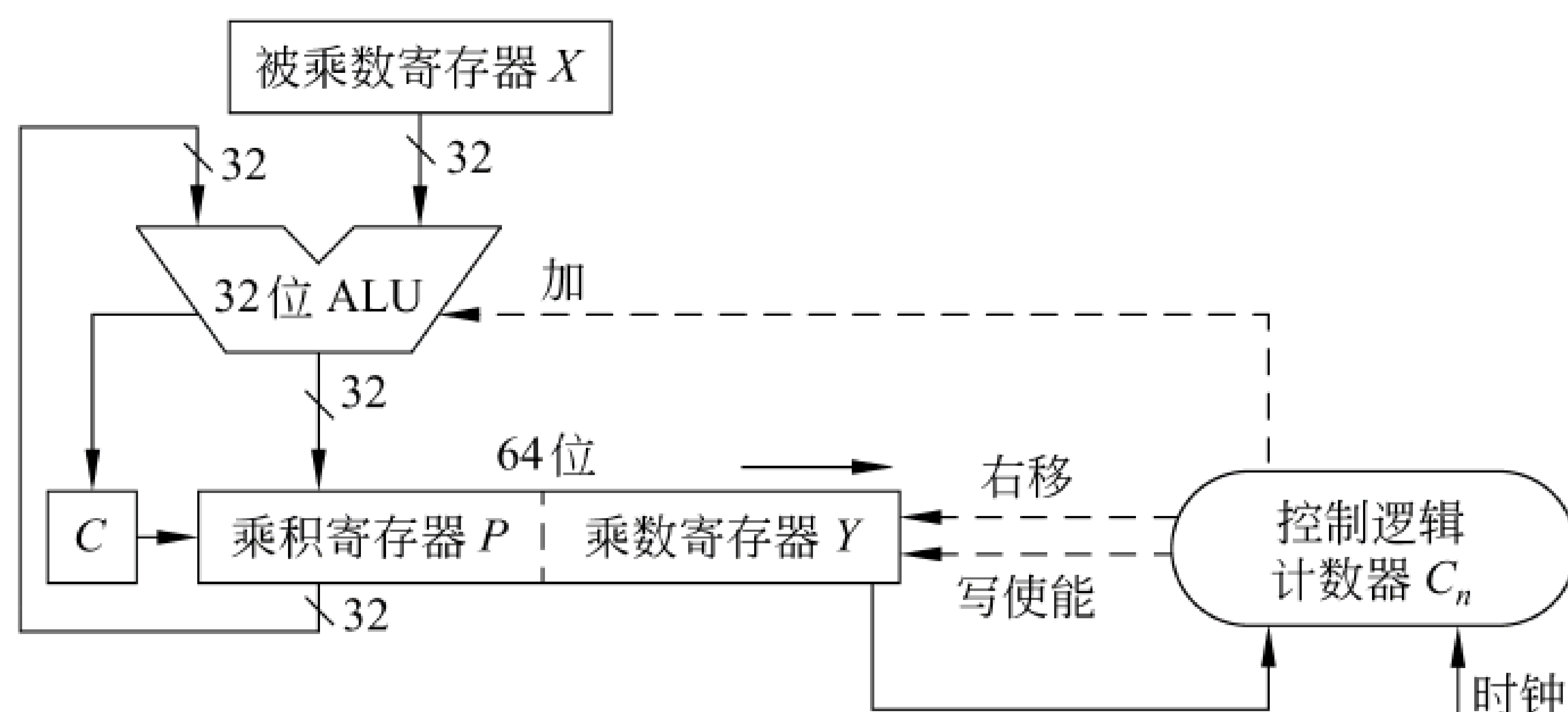


图 3.19　实现 32 位无符号数乘法运算的逻辑结构图

图 3.19 中各部件的功能说明如下。

被乘数寄存器 X：存放被乘数。

乘积寄存器 P：开始时，置初始部分积 $P_0=0$；结束时，存放的是 64 位乘积的高 32 位。

乘数寄存器 Y：开始时，置乘数；结束时，存放的是 64 位乘积的低 32 位。

进位触发器 C：保存加法器的进位信号。

计数器 C_n：存放循环次数。初值是 32，每循环一次，C_n 减 1，当 $C_n=0$ 时，乘法运算结束。

ALU：乘法核心部件。在控制逻辑控制下，对乘积寄存器 P 和被乘数寄存器 X 的内容进行"加"运算，在"写使能"控制下运算结果被送回乘积寄存器 P，进位位存放在 C 中。

每次循环都要对进位位 C、乘积寄存器 P 和乘数寄存器 Y 实现同步"右移"，此时，进位信号 C 移入寄存器 P 的最高位，寄存器 P 的最低位移出到寄存器 Y 的最高位，寄存器 Y 的最低位移出，0 移入进位位 C 中。从最低位 y_n 开始，逐次把乘数的各个数位 y_{n-i} 移到寄存器 Y 的最低位上。因此，寄存器 Y 的最低位被送到控制逻辑以决定被乘数是否"加"到部分积上。图 3.20 是无符号数乘法的流程图。

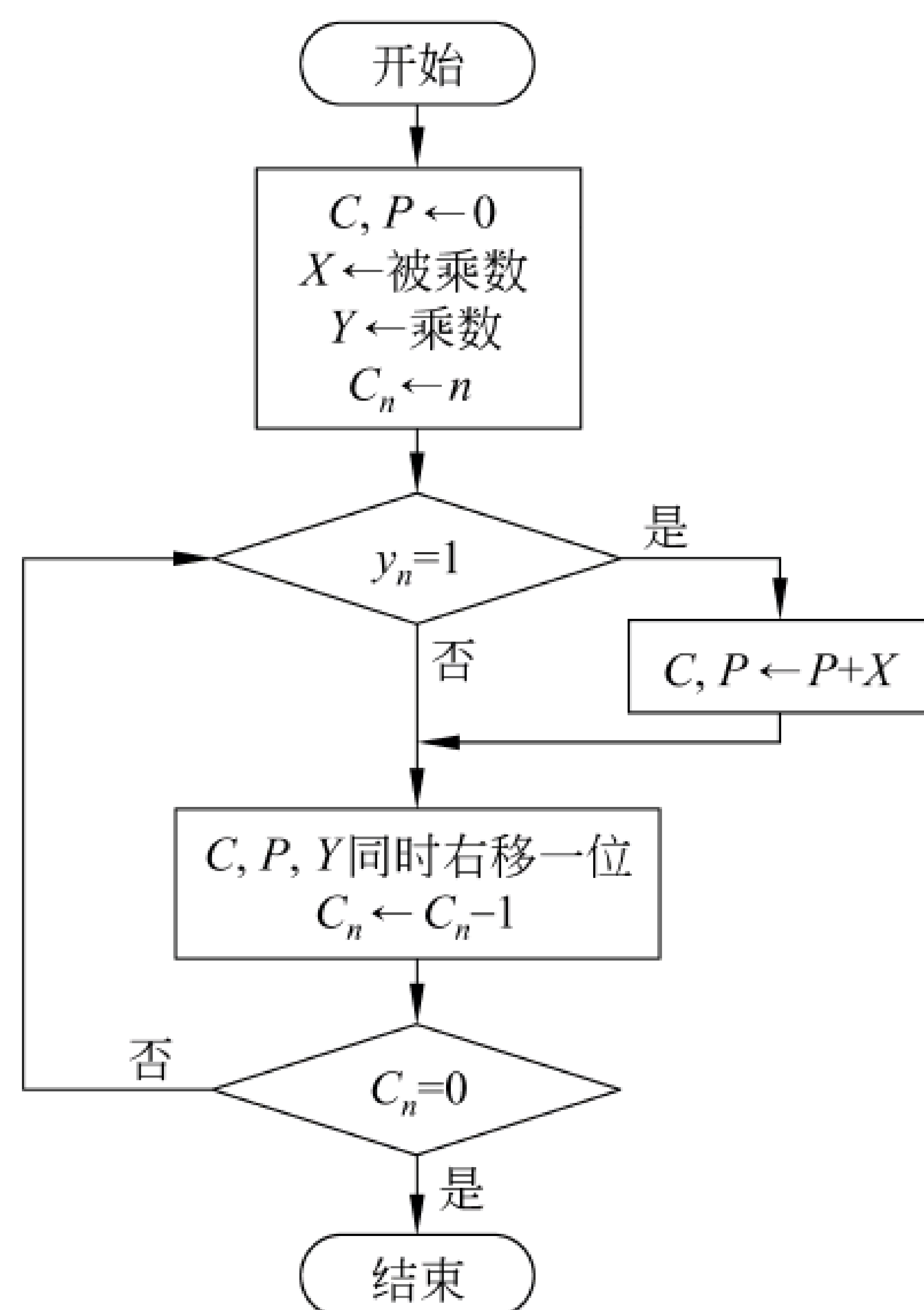


图 3.20　无符号数乘法操作流程

对于原码定点小数的乘法运算，只要根据上述无符号数的乘法运算得到乘积的数值部分，然后再加上符号位，就可以得到最终原码表示的乘积。需要补充说明一点，当被乘数或乘数中至少有一个为全0时，结果直接得0，不再进行乘法运算。

例 3.7 已知 $[X]_{原}=0.1101$，$[Y]_{原}=0.1011$，用原码一位乘法计算 $[X\times Y]_{原}$。

解：先采用无符号数乘法计算 1101×1011 的乘积，原码一位乘法过程如下。

C	P	Y	说明
0	0 0 0 0	1 0 1 1	$P_0=0$
	+1 1 0 1		$y_4=1, +X$
0	1 1 0 1		C, P 和 Y 同时右移一位
0	0 1 1 0	1 | 1 0 1	得 P_1
	+1 1 0 1		$y_3=1, +X$
1	0 0 1 1		C, P 和 Y 同时右移一位
0	1 0 0 1	1 1 | 1 0	得 P_2
	+0 0 0 0		$y_2=0$, 不作加法(加0)
0	1 0 0 1	1 1 | 1 0	C, P 和 Y 同时右移一位
0	0 1 0 0	1 1 1 | 1	得 P_3
	+1 1 0 1		$y_1=1, +X$
1	0 0 0 1		C, P 和 Y 同时右移一位
0	1 0 0 0	1 1 1 1 |	得 P_4

符号位为 $0\oplus 0=0$，因此，$[X\times Y]_{原}=0.10001111$。

*2. 原码二位乘法

对于 n 位原码一位乘法来说，需要经过 n 次“判断—加法—右移”循环，运算速度较慢。如果对乘数的每两位取值情况进行判断，使每步求出对应于该两位的部分积，则可将乘法速度提高一倍。这种方法被称为原码二位乘法，只需在原码一位乘法的基础上增加少量的逻辑线路，就可实现原码二位乘法。

考察乘数每两位的组合以及对应的求部分积的操作情况，归纳如下：

$$00\text{——}P_{i+1}=2^{-2}(P_i+0)$$

$$01\text{——}P_{i+1}=2^{-2}(P_i+X)$$

$$10\text{——}P_{i+1}=2^{-2}(P_i+2X)$$

$$11\text{——}P_{i+1}=2^{-2}(P_i+3X)$$

对于上述“+0”和“$+X$”的情况，与前面原码一位乘法一样即可；对于“$+2X$”，可通过 X 左移1位来实现；对于“$+3X$”的实现则有两种方法：(1)分“$+X$”和“$+2X$”两次进行，需做两次加法。(2)以 $4X-X$ 代替 $3X$，在本次运算中只执行 $-X$，而 $+4X$ 则延迟到下一次执行。这种情况下，部分积 $P_{i+1}=2^{-2}(P_i+3X)=2^{-2}(P_i-X+4X)=2^{-2}(P_i-X)+X$。“$-X$”用 $+[-X]_{补}$ 实现。因为下一次部分积已右移了两位，所以上次未完成的“$+4X$”已变成“$+X$”。可用一个触发器 T 记录是否需要在下次执行“$+X$”，若是，则 $1\rightarrow T$。因此，实际操作中用 y_{i-1}、y_i 和 T 三位来控制乘法操作，运算规则如表 3.3 所示。

表 3.3 原码两位乘法运算规则

y_{i-1}	y_i	T	操　　作	迭代公式
0	0	0	$0 \to T$	$2^{-2}(P_i)$
0	0	1	$+X \quad 0 \to T$	$2^{-2}(P_i+X)$
0	1	0	$+X \quad 0 \to T$	$2^{-2}(P_i+X)$
0	1	1	$+2X \quad 0 \to T$	$2^{-2}(P_i+2X)$
1	0	0	$+2X \quad 0 \to T$	$2^{-2}(P_i+2X)$
1	0	1	$-X \quad 1 \to T$	$2^{-2}(P_i-X)$
1	1	0	$-X \quad 1 \to T$	$2^{-2}(P_i-X)$
1	1	1	$1 \to T$	$2^{-2}(P_i)$

因为原码乘法是对数值部分(可看成无符号数或带符号正数)执行乘法运算,而正数的补码等于正数本身。所以,对于正数的运算可以采用补码运算方式。因此,这里的加减运算采用补码加减方式,故需在无符号数前添一位符号位 0;此外,部分积可能与 $2X$ 相加,因此存在将加数左移一位的操作,左移后的数值部分会进到符号位,因而需要两位符号位;左移后的被乘数再与部分积相加时,得到的和的数值部分又可能会进到前面一位符号位上。为此,在原码两位乘法运算中采用 3 位符号位,以模 8 补码形式操作。

例 3.8 已知 $[X]_{原}=0.111001$,$[Y]_{原}=0.100111$,用原码两位乘法计算$[X\times Y]_{原}$。

解:先采用无符号数乘法计算 111001×100111 的乘积,原码两位乘法过程如下:

$[|X|]_{补}=000\ 111001$,$[|2X|]_{补}=001\ 110010$,$[-|X|]_{补}=111\ 000111$。

P	Y	T	说明
000 000000	100111	0	$P_0=0$, $T=0$
+111 000111			$y_5y_6T=110$, $-X$, $T=1$
111 000111			P 和 Y 同时右移 2 位
111 110001	11 1001	1	得 P_1
+001 110010			$y_3y_4T=011$, $+2X$, $T=0$
001 100011			P 和 Y 同时右移 2 位
000 011000	1111 10	0	得 P_2
+001 110010			$y_1y_2T=100$, $+2X$, $T=0$
→ 010 001010			P 和 Y 同时右移 2 位
000 100010	101111	0	得 P_3

加上符号位,得 $[X\times Y]_{原}=0.100010101111$。

在上述例子中,计算 P_3 时,加法溢出使数值部分占用了两位符号位,如箭头所指处。若采用模 4 补码运算,则在进行 P 和 Y 同时右移两位操作时,按照补码右移规则,得到的 P_3 是负数。显然这是错误的,因为两个正数相乘,乘积不可能是负数。因此在此必须采用模 8 补码运算。

上述例子是一个 6 位无符号数乘法运算,只用了三次循环。很明显,两位乘法相对于一位乘法来说,运算速度加快了一倍。

3.3.5 补码乘法运算

补码作为机器中带符号整数的表示形式，需要计算机能实现定点补码整数的乘法运算。根据每次部分积是一位相乘得到还是两位相乘得到，有补码一位乘法和补码两位乘法。

1. 补码一位乘法

原码乘法运算需要将符号位和数值部分分开来运算。对于补码乘法运算，同样也可以先将补码整数转换为原码整数，再将符号位和数值部分分开来运算，最后再将乘积转换为补码表示。但是，这样做会增加许多数据转换的开销，降低了乘法运算的速度。

补码加减法运算可以让符号位与数值位一起参与运算，且结果的符号位也在运算过程中产生。同样，补码乘法也可以将符号位和数值部分合在一起进行运算。

从下面两个例子出发，来看补码乘法是否可将符号位和数值部分合在一起进行运算。

例如，假定 $X=+1011$，$Y=+0001$，那么 $[X]_{补}=0\ 1011$，$[Y]_{补}=0\ 0001$，用 X 和 Y 相乘然后求补码，得：$[X\times Y]_{补}=0\ 00001011$；用$[X]_{补}$和$[Y]_{补}$直接相乘，得：$[X]_{补}\times[Y]_{补}=000001011$，显然，在这个例子中，存在关系：$[X\times Y]_{补}=[X]_{补}\times[Y]_{补}$。

又例如，假定 $X=+1011$，$Y=-0001$，那么 $[X]_{补}=0\ 1011$，$[Y]_{补}=1\ 1111$，用 X 和 Y 相乘然后求补码，得：$[X\times Y]_{补}=1\ 11110101$；用$[X]_{补}$和$[Y]_{补}$直接相乘，得：$[X]_{补}\times[Y]_{补}=1\ 01010101$，显然，在这个例子中，上述关系不存在，即$[X\times Y]_{补}\neq[X]_{补}\times[Y]_{补}$。

从上面两个例子可看出：两个正数补码的乘积等于乘积的补码，否则这种关系不成立。

A. D. Booth 提出了一种补码相乘算法，可以将符号位与数值位合在一起参与运算，直接得出用补码表示的乘积，且正数和负数同等对待。这种算法被称之为 Booth（布斯）乘法。

因为补码用来表示带符号整数，机器字长都是字节的倍数，所以，我们考察偶数位的补码定点整数的乘法运算。也即假定$[X]_{补}$和$[Y]_{补}$是两个偶数位的补码定点整数，$[X\times Y]_{补}$的 Booth 乘法递推公式推导如下。

设$[X]_{补}=x_{n-1}\cdots x_1x_0$，$[Y]_{补}=y_{n-1}\cdots y_1y_0$，根据补码定义，可得到 Y 的真值的计算公式如下。

$$
\begin{aligned}
Y &= -y_{n-1}2^{(n-1)}+\sum_{i=0}^{n-2}y_i2^i \\
&= -y_{n-1}2^{(n-1)}+y_{n-2}2^{(n-2)}+\cdots+y_12^1+y_02^0 \\
&= -y_{n-1}2^{(n-1)}+y_{n-2}2^{(n-1)}-y_{n-2}2^{(n-2)}+\cdots+y_12^2-y_12^1+y_02^1-y_02^0 \\
&= (y_{n-2}-y_{n-1})2^{(n-1)}+(y_{n-3}-y_{n-2})2^{(n-2)}+\cdots+(y_0-y_1)\ 2^1+(0-y_0)2^0 \\
&= \sum_{i=0}^{n-1}(y_{i-1}-y_i)2^i
\end{aligned}
$$

这里假设 $y_{-1}=0$。因此，

$$
[X\times Y]_{补}=[X\times\sum_{i=0}^{n-1}(y_{i-1}-y_i)\ 2^i]_{补} \tag{3-8}
$$

与推导无符号数乘法算法一样，可以不考虑小数点位置，只要最终的乘积约定好小数点位置就可以了。因此，公式(3-8)的右边可以通过乘以 2^{-n}来变换成以下形式：

$$
\left[X\times\sum_{i=0}^{n-1}(y_{i-1}-y_i)2^{-(n-i)}\right]_{补} \tag{3-9}
$$

将式(3-9)展开后得到如下递推公式：

$$[P_{i+1}]_{补} = [2^{-1}(P_i + (y_{i-1} - y_i)X)]_{补} \quad (i = 0,1,2,\cdots,n-1) \tag{3-10}$$

此公式中 P_i 为上次部分积，P_{i+1} 为本次部分积。令$[P_0]_{补}=0$，则有：

$$\left.\begin{aligned} [P_1]_{补} &= [2^{-1}(P_0 + (y_{-1} - y_0) \times X)]_{补} \\ &\vdots \\ [P_{n-1}]_{补} &= [2^{-1}(P_{n-2} + (y_{n-3} - y_{n-2}) \times X)]_{补} \\ [P_n]_{补} &= [2^{-1}(P_{n-1} + (y_{n-2} - y_{n-1}) \times X)]_{补} \end{aligned}\right\} \tag{3-11}$$

比较式(3-8)和式(3-11)，可以得出结论：$[X\times Y]_{补}=2^n[P_n]_{补}$，因此，只要将最终部分积$[P_n]_{补}$的小数点约定到最右边就行了。

由递推公式(3-10)可以知道，在求得$[P_i]_{补}$后，根据对乘数中连续两位 y_iy_{i-1} 的判断，就可求得$[P_{i+1}]_{补}$。

若 $y_i\,y_{i-1}=01$，则$[P_{i+1}]_{补}=[2^{-1}(P_i+X)]_{补}$。

若 $y_i\,y_{i-1}=10$，则$[P_{i+1}]_{补}=[2^{-1}(P_i-X)]_{补}$。

若 $y_i\,y_{i-1}=00$ 或 11，则$[P_{i+1}]_{补}=[2^{-1}(P_i+0)]_{补}$。

上述式子$[2^{-1}(P_i\pm X)]_{补}$可通过执行$[P_i]_{补}+[\pm X]_{补}$后右移一位实现。此时，采用的是补码右移方式，即是带符号数的算术右移。

根据上述分析，归纳出补码乘法运算规则如下：

(1) 乘数最低位增加一位辅助位 $y_{-1}=0$。

(2) 判断 $y_i\,y_{i-1}$ 的值，决定是 $+X$、$-X$、$+0$。

(3) 每次加减后，算术右移一位，得到部分积。

(4) 重复第(2)和第(3)步 n 次，结果得$[X\times Y]_{补}$。

布斯乘法的算法流程如图 3.21 所示。

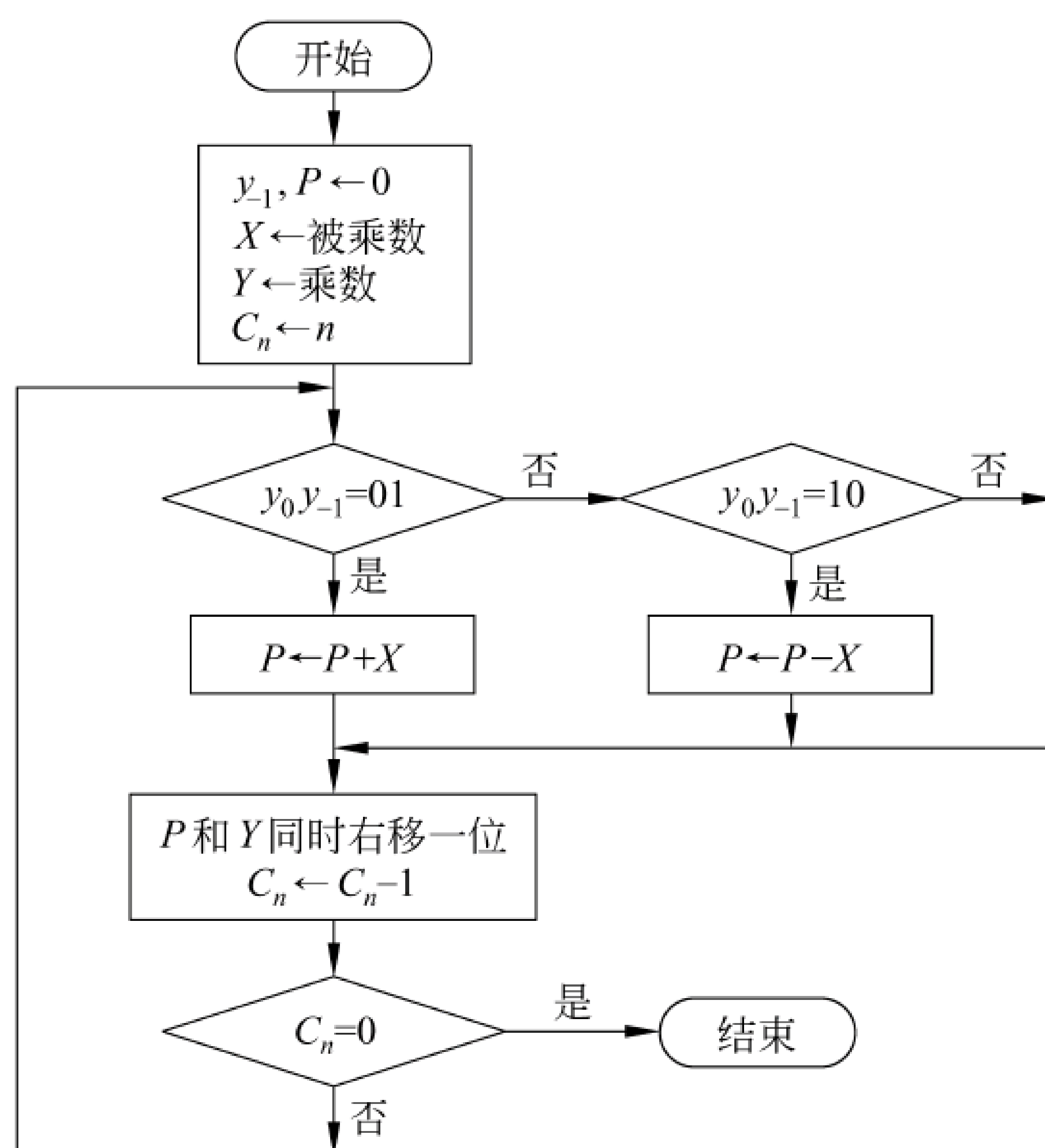


图 3.21　布斯乘法运算流程图

图 3.22 是实现 32 位补码一位乘法的逻辑结构图，与图 3.19 所示的无符号数乘法的逻辑结构类似，只是部分控制逻辑不同。

补码一位乘法要点如下：

(1) 初始化时，在乘数后面一位 y_{-1} 补 0。

(2) 将乘积最低两位 y_0y_{-1} 取到控制逻辑，以判断执行 $+X$、$-X$、$+0$ 操作。

(3) 在控制逻辑控制下，ALU 可能要执行“补码加”或“补码减”运算。

(4) ALU 运算后的进位位不需要保留。

(5) 右移时采用算术右移方式。

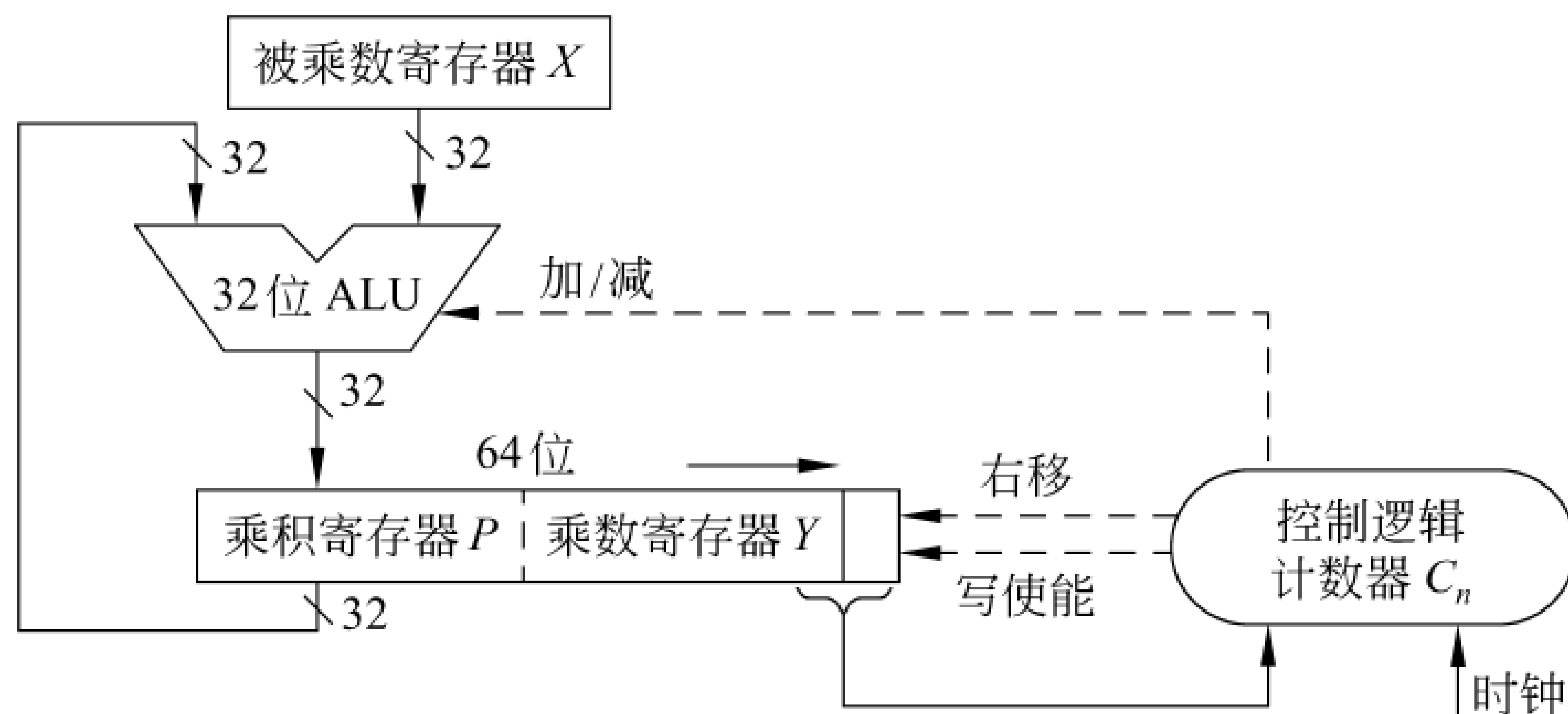


图 3.22 实现补码一位乘法的逻辑结构图

例 3.9 已知 $[X]_补=1\ 101$，$[Y]_补=0\ 110$，要求用布斯乘法计算 $[X\times Y]_补$。

解：$[-X]_补=0\ 011$，布斯乘法过程如下：

P	Y	y_{-1}	说明
0000	0110	0	设 $y_{-1}=0$, $[P_0]_补=0$
			$y_0y_{-1}=00$, P、Y直接右移一位
0000	0011	0	得 $[P_1]_补$
+0011			$y_1y_0=10$, $+[-X]_补$
0011			P、Y同时右移一位
0001	1001	1	得 $[P_2]_补$
			$y_2y_1=11$, P、Y直接右移一位
0000	1100	1	得 $[P_3]_补$
+1101			$y_3y_2=01$, $+[X]_补$
1101			P, Y同时右移一位
1110	1110	0	得 $[P_4]_补$

因此，$[X\times Y]_补=1110\ 1110$。

验证：$X=-011B=-3$，$Y=+110B=6$，$X\times Y=-0001\ 0010B=-18$，结果正确。

布斯乘法的算法过程为 n 次“判断—加减—右移”循环，从流程图可以看出，在布斯乘法中，遇到连续的 1 或连续的 0 时，可跳过加法运算直接进行右移操作，因此，布斯算法的运算效率较高。

*2. 补码两位乘法

补码乘法也可以采用两位一乘的方法，把乘数分成两位一组，根据两位代码的组合决定加减被乘数的倍数，形成的部分积每次右移两位。补码两位一乘的方法可以用布斯乘法过程来推导。

假设用布斯乘法已经求得部分积$[P_i]_补$，则部分积$[P_{i+1}]_补$和$[P_{i+2}]_补$可分别写为：

$$[P_{i+1}]_补 = 2^{-1}([P_i]_补 + (y_{i-1} - y_i)[X]_补) \tag{3-12}$$

$$[P_{i+2}]_补 = 2^{-1}([P_{i+1}]_补 + (y_i - y_{i+1})[X]_补) \tag{3-13}$$

把式(3-12)代入上式(3-13)中，可以得到以下表达式。

$$\begin{aligned}[P_{i+2}]_补 &= 2^{-1}(2^{-1}([P_i]_补 + (y_{i-1} - y_i)[X]_补) + (y_i - y_{i+1})[X]_补) \\ &= 2^{-2}([P_i]_补 + (y_{i-1} + y_i - 2y_{i+1})[X]_补)\end{aligned} \tag{3-14}$$

从(3-14)可看出，由乘数中相邻三位代码 y_{i+1}、y_i、y_{i-1}的值的组合作为判断依据，可以跳过$[P_{i+1}]_补$的计算步骤，即从$[P_i]_补$直接求得$[P_{i+2}]_补$。乘数3位代码y_{i+1}、y_i、y_{i-1}构成的判断规则如表3.4所示。

表3.4 补码两位乘法判断规则

y_{i+1}	y_i	y_{i-1}	操 作	迭代公式
0	0	0	+0	$2^{-2}[P_i]_补$
0	0	1	$+[X]_补$	$2^{-2}\{[P_i]_补+[X]_补\}$
0	1	0	$+[X]_补$	$2^{-2}\{[P_i]_补+[X]_补\}$
0	1	1	$+2[X]_补$	$2^{-2}\{[P_i]_补+2[X]_补\}$
1	0	0	$+2[-X]_补$	$2^{-2}\{[P_i]_补+2[-X]_补\}$
1	0	1	$+[-X]_补$	$2^{-2}\{[P_i]_补+[-X]_补\}$
1	1	0	$+[-X]_补$	$2^{-2}\{[P_i]_补+[-X]_补\}$
1	1	1	+0	$2^{-2}[P_i]_补$

补码两位乘法运算过程与布斯乘法相似，因此称为改进布斯算法(Modified Booth Algorithm，MBA)，也称为基4(Radix-4)布斯算法。该算法通过对乘数按两位编码可将部分积的数目压缩一半，从而提高运算速度。

从表3.4可看出，所需加减运算有$+[X]_补$、$+2[X]_补$、$+2[-X]_补$、$+[-X]_补$四种情况。$+2[X]_补$、$+2[-X]_补$采用将被乘数左移一位后和部分积进行加减的方法实现。因为左移一位后再和部分积加减，其数值部分可能进到符号位，所以需要使用两位符号位。每次部分积和乘数共同右移两位。初始时，置附加位y_{-1}为0，乘积寄存器最高位前面添加一位附加符号位0。最终的乘积高位部分在乘积寄存器P中，低位部分在乘数寄存器Y中。

因为字长总是8的倍数，所以补码的位数n应该是偶数，因此，总循环次数为$n/2$。

例3.10 已知$[X]_补$=1 101，$[Y]_补$=0 110，要求用补码两位乘法计算$[X\times Y]_补$。

解： $[-X]_补$=0 011，用补码二位乘法计算$[X\times Y]_补$的过程如下。

P_n	P	Y	y_{-1}	说　明
0	0000	0110	0	设 $y_{-1}=0$, $[P_0]_补=0$
+0	0110			$y_1y_0y_{-1}=100$, $+2[-X]_补$
0	0110			P 和 Y 同时右移二位
0	0001	1001	1	得 $[P_2]_补$
+1	1010			$y_3y_2y_1=011$, $+2[X]_补$
1	1011			P 和 Y 同时右移二位
1	1110	1110		得 $[P_4]_补$

因此 $[X\times Y]_补=1110\ 1110$，与一位补码乘法(布斯乘法)所得结果相同，但循环次数减少了一半。

*3.3.6　快速乘法器

乘法是数字信号处理中重要的基本运算。在图像、语音、加密等数字信号处理领域，乘法器扮演着重要的角色，并在很大程度上决定着系统性能。乘法器也是处理器中进行数据处理的关键部件，大约 1/3 是乘法运算。因此，有必要考虑实现高速乘法运算。前面介绍的原码两位乘法和补码两位乘法(MBA)，通过一次判断两位乘数来提高乘法速度。同理，可以采用一次判断更多位乘数的乘法，但是多位乘法运算的控制复杂度呈几何级数增长，实现难度很大。随着大规模集成电路技术的飞速发展，出现了采用硬件叠加或流水处理的快速乘法器件，如阵列乘法器就是其中之一。

图 3.23 是用手算进行两个 4 位无符号数相乘的示意图。在手算算式中，每个 x_iy_j($i=1\sim4$，$j=1\sim4$)都是由两个 1 位的二进制数相乘得到的。第 1 行为 x_iy_4($i=1\sim4$)，第 2 行为 x_iy_3($i=1\sim4$)，第 3 行为 x_iy_2($i=1\sim4$)，第 4 行为 x_iy_1($i=1\sim4$)，所以，每个 x_iy_j($i=1\sim4$，$j=1\sim4$)可以用一个"与"门实现。每行都向左错一位，最终将权相等的位的积相加，形成最终的乘积 $P=P_7P_6P_5P_4P_3P_2P_1P_0$。

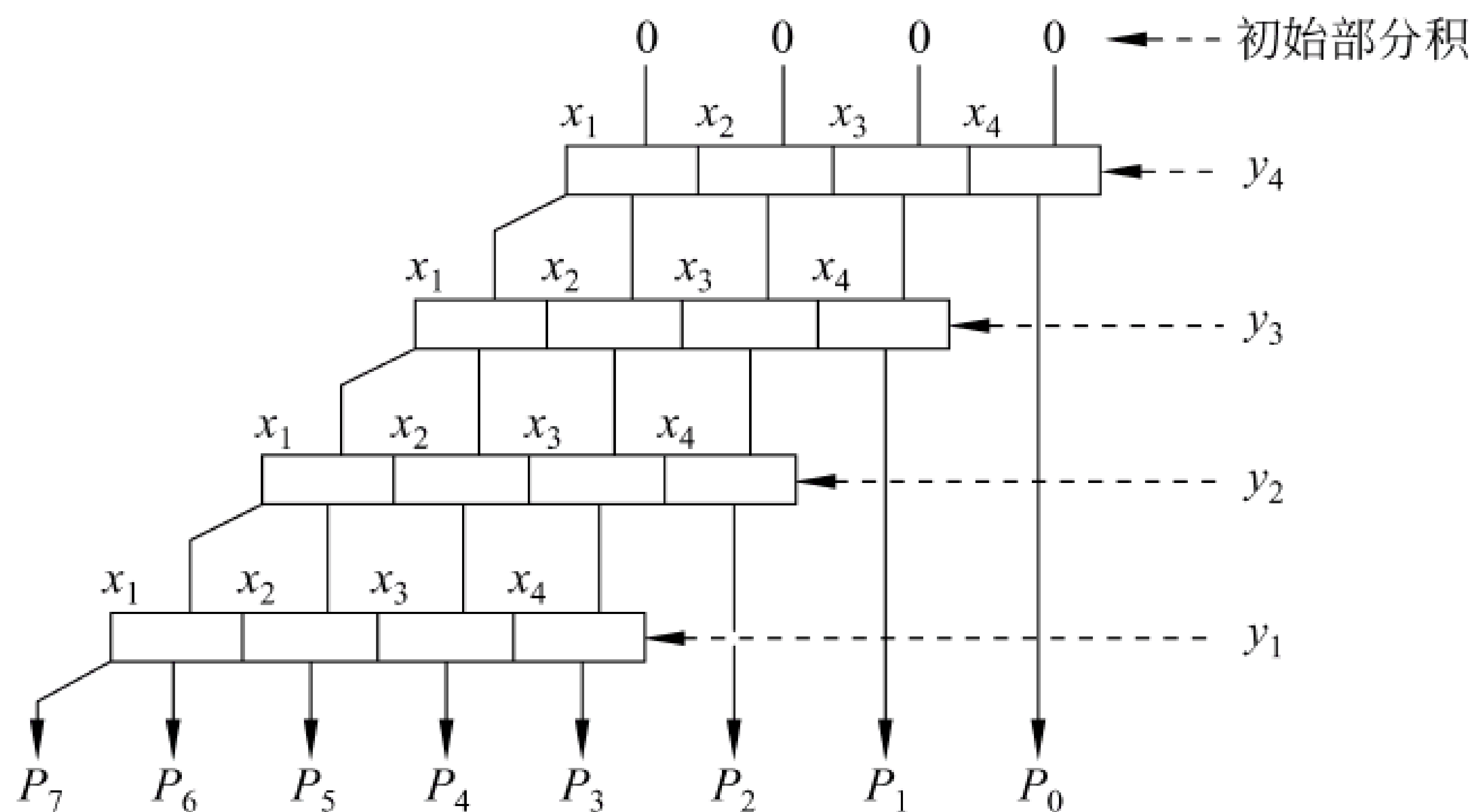


图 3.23　4 位无符号数的手算过程

在计算机内，用组合逻辑线路可以构成一个实现上述执行过程的乘法器。如图 3.24 所示，该乘法器为阵列结构形式，故称为阵列乘法器(Array Multiplier)。图中实现了 $X\times Y$，其中 $X=x_1x_2x_3x_4$，$Y=y_1y_2y_3y_4$。X 和 Y 是无符号数。一位乘积 x_iy_j 可以用一个两输入端的"与"门实现。每一次加法操作用一个全加器实现。2^i 和 2^j 的因子所蕴含的移位由全加

器的空间错位来实现。与门和全加器的功能可用一个单元组合起来，称为一个细胞模块。在图中用一个方框来表示。

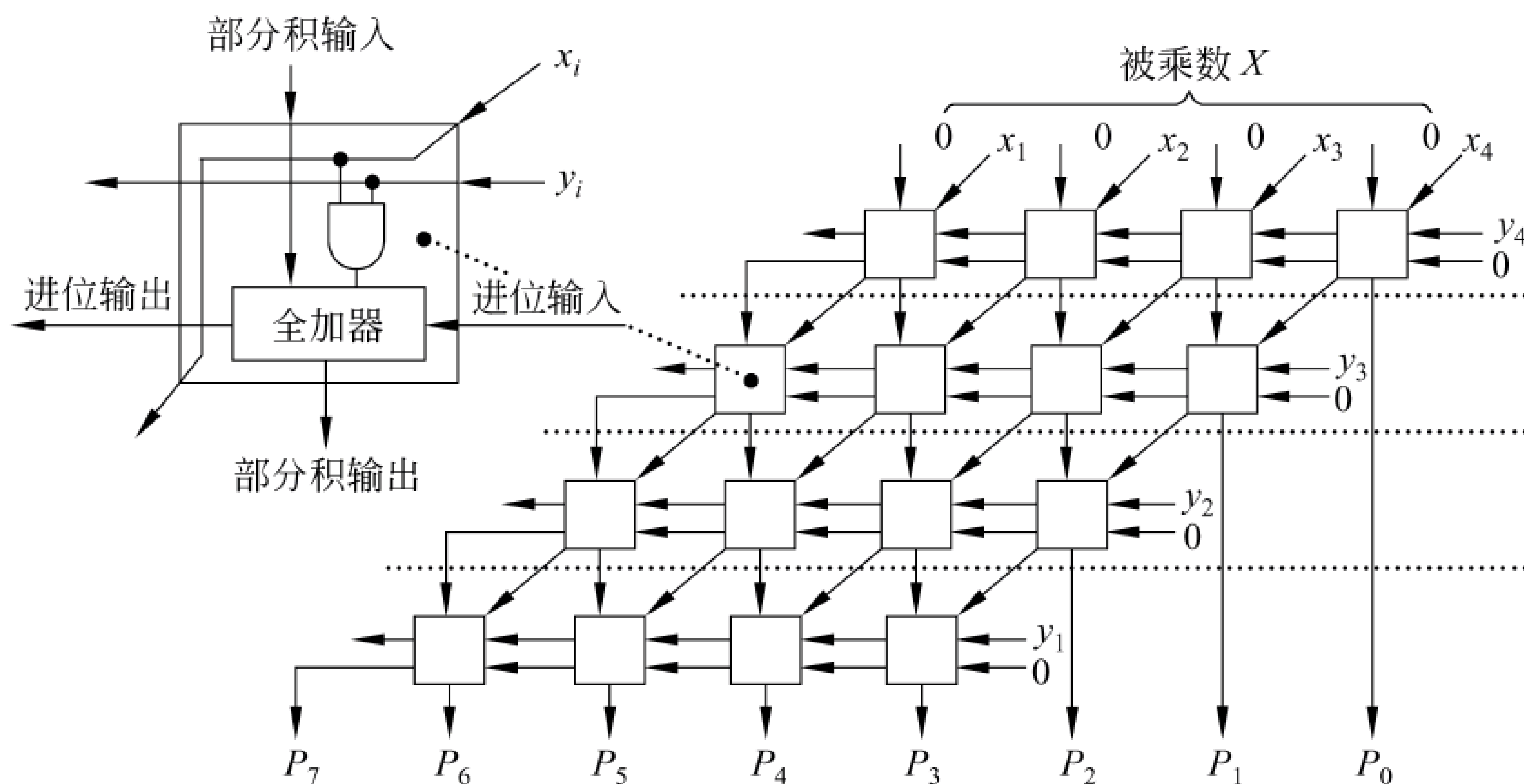


图 3.24　4×4 位基于 CRA 的阵列乘法器

阵列乘法器基于移位与求和算法实现，每一行被乘数与乘数中的某一位相乘，产生一组部分积。即每一行由乘数的每一数位 $y_j(j=1,2,3,4)$ 控制得到本级的部分积 $x_i2^{(4-i)}\times y_j$ $(i=1,2,3,4)$。而每一斜列则由被乘数的每一数位 $x_i(i=1,2,3,4)$ 控制，即为 $x_i\times y_j2^{(4-j)}$ $(j=1,2,3,4)$。如此求出全部部分积，最后对所有部分积求和得到乘积，整个电路的延迟取决于用于求和的加法阵列结构。

图 3.24 中采用的是基于行波进位加法器(Carry Ripple Adder，CRA)的阵列乘法器，采用串行进位，每一级部分积的生成不仅依赖上一级的部分积，还依赖于上一级的最终进位，因而运算速度慢。为加快运算速度，加法阵列可改用基于 CSA(Carry Save Adder，进位保留加法器)方式的结构，如图 3.25 所示。CSA 将本级进位与本级和一样同时输出至下一级，而不是向前传递到本级的下一位，因而求和速度快，且向下级传递的速度与字长无关。

阵列乘法器结构规范，标准化程度高，有利于布局布线，适合用超大规模集成电路实现，且能获得较高的运算速度，其乘法速度仅取决于逻辑门和加法器的传输延迟，随着集成电路价格的不断下降，阵列乘法器在某些数字系统中也被大量使用，例如在数字信号处理系统中受到重视。

阵列乘法器至少要做 $O(N)$ 次加法，速度较慢。为了进一步提高速度，部分积求和电路可采用树形结构。树型结构可以减少求和级数，是提高乘法运算速度的一种方法。1961 年 Wallac 提出的华莱士树(Wallace Tree，WT)结构是其中最著名的一种，它对 16 位以上的乘法运算尤其适用。WT 结构将全部部分积按列分组，每列对应一组加法器，各列同时相加，前一列进位传至后一列，生成新的部分积阵列；按同样的方法化简新的阵列，直至只剩两行部分积，最后用高速加法器求和得到最终乘积。WT 结构只需做 $O(\log N)$ 次加法，因而运算速度快。可将改进布斯乘法 MBA 和 WT 结合起来进一步加快乘法速度，MBA 用来减少部分积个数，而 WT 用来缩短部分积求和时间。

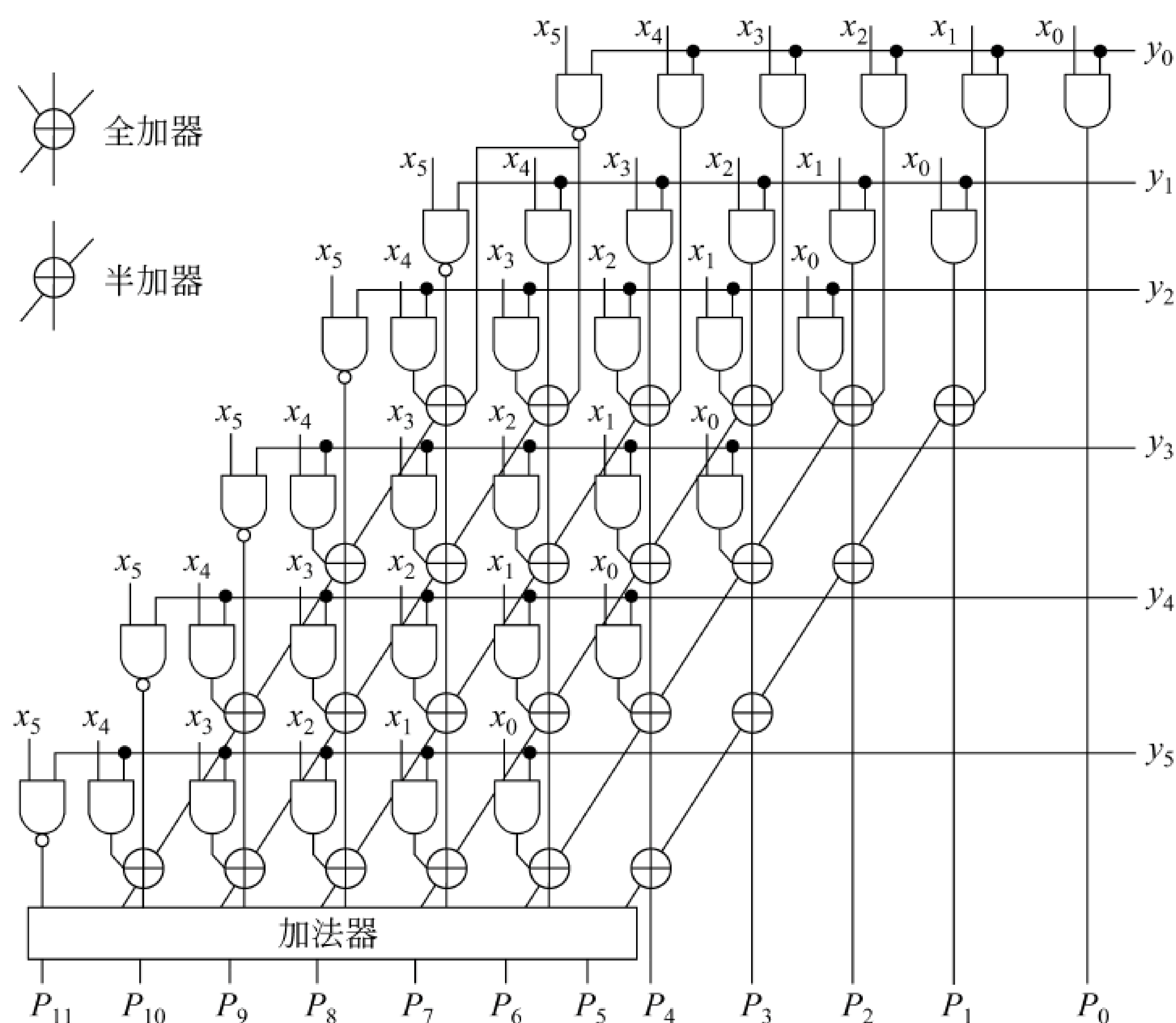


图 3.25　6×6 位基于 CSA 的阵列乘法器

3.3.7　原码除法运算

在进行定点数除法运算前，首先要对被除数和除数的取值和大小进行相应的判断，以确定除数是否为 0、商是否为 0、是否溢出或为不确定的值 NaN。通常的判断操作如下：

(1) 若被除数为 0、除数不为 0，或者定点整数除法时 |被除数| < |除数|，则说明商为 0，不再继续执行。

(2) 若被除数不为 0、除数为 0，则发生“除数为 0”异常。

(3) 若被除数和除数都为 0，则有些机器产生一个不发信号的 NaN，即 quiet NaN。

只有当被除数和除数都不为 0，并且商也不可能为 0 时，才进一步进行除法运算。

原码作为浮点数尾数的表示形式，需要计算机能实现定点原码小数的除法运算。除法运算与乘法运算很相似，都是一种移位和加减运算的迭代过程，但比乘法运算更加复杂。下面以两个定点正数为例，说明手算除法步骤。

假定被除数 $X=10011101$，除数 $Y=1011$，以下是这两个数相除的手算过程。

```
                 1 1 1 0      商
除数 1011 ) 1 0 0 1 1 1 0 1   被除数
            1 0 1 1
            ---------------
              1 0 0 0 1     ←┐
                1 0 1 1      │
              -----------    │
                0 1 1 0 0   ←┤ 中间余数
                  1 0 1 1    │
                ---------    │
                  0 0 1 1   ←┘
                  0 0 0 0
                ---------
                  0 0 1 1     余数
```

从上述过程和结果来看，手算除法的基本要点如下。

(1) 被除数与除数相减，若够减，则上商为1；若不够减，则上商为0。

(2) 每次得到的差为中间余数，将除数右移后与上次的中间余数比较。用中间余数减除数，若够减，则上商为1；若不够减，则上商为0。

(3) 重复执行第(2)步，直到求得的商的位数足够为止。

计算机内部的除法运算与手算算法一样，通过被除数(中间余数)减除数来得到每一位的商。

原码除法运算与原码乘法运算一样，要将符号位和数值位分开来处理。商的符号为相除两数符号的"异或"值，商的数值为两数绝对值之商。因此，以下考虑定点正整数和定点正小数的除法运算。除法逻辑结构类似于乘法逻辑结构，如图3.26所示是一个32位除法逻辑结构示意图。

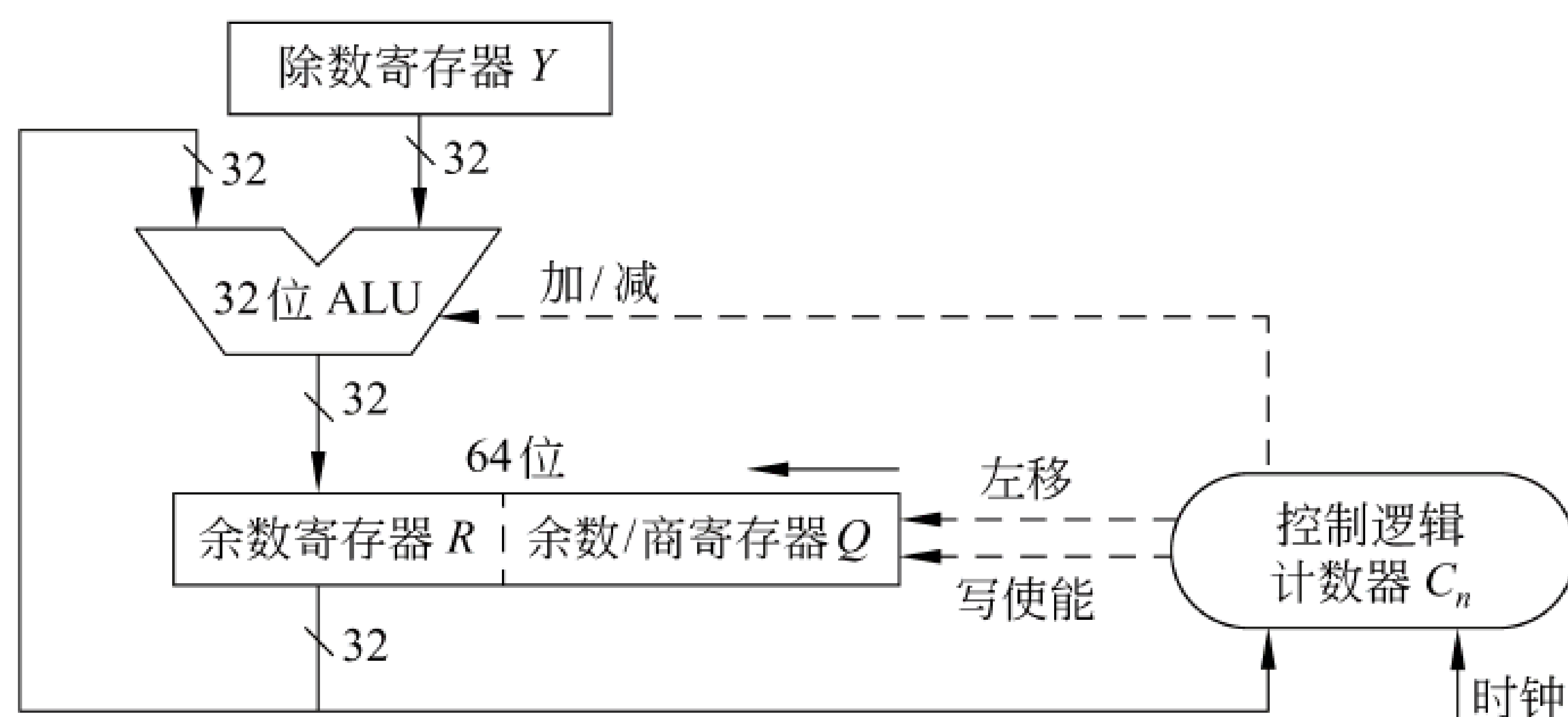


图3.26　32位除法运算逻辑结构

图3.26中各部件的功能说明如下。

除数寄存器 Y：存放除数。

余数寄存器 R：开始时，将被除数的高32位置于此，作为初始中间余数 R_0 的高位部分；结束时，存放的是余数。

余数/商寄存器 Q：开始时，将低32位被除数置于此，作为初始中间余数 R_0 的低位部分；结束时，存放的是32位商。寄存器 Q 中存放的并不是商的全部位数，而是部分为被除数或中间余数，部分为商，只有到最后一步才是商的全部位数。

计数器 C_n：存放循环次数。初值是32，每循环一次，C_n 减1，当 $C_n=0$ 时，除法运算结束。

ALU：除法器核心部件。在控制逻辑控制下，对于余数寄存器 R 和除数寄存器 Y 的内容进行"加"或"减"运算，在"写使能"控制下运算结果被送回余数寄存器 R。

每次循环都要对余数寄存器 R 和余数/商寄存器 Q 实现同步"左移"，左移时，寄存器 Q 的最高位移入寄存器 R 的最低位，寄存器 Q 中空出的最低位上被上"商"。从低位开始，逐次把商的各个数位左移到寄存器 Q 中。每次由控制逻辑根据ALU运算结果的符号位来决定上商为0还是1。

由图3.26可知，两个32位数相除，必须把被除数扩展成一个64位数。推而广之，n 位定点数的除法，实际上是用一个 $2n$ 位的数去除以一个 n 位的数，得到一个 n 位的商。因此需要进行被除数的扩展。

定点正整数和定点正小数的除法运算，都可以用上述图3.26所示的除法逻辑来实现。只是被除数扩展的方法不太一样，此外，导致溢出的情况也有所不同。

(1) 对于两个 n 位定点正整数相除的情况，也即当两个 n 位无符号数相除时，只要将被除数 X 的高位添 n 个0即可。即 $X=x_{n-1}x_{n-2}\cdots x_1x_0$ 变成 $X=00\cdots00\ x_{n-1}x_{n-2}\cdots x_1x_0$。显然，对被除数预置时，$R$ 寄存器中为全0，Q 寄存器中为被除数 X。这种方式通常称为单精度除法，其商的位数一定不会超过 n 位，因此不会发生溢出。

(2) 对于两个 n 位定点正小数相除的情况，也即当两个作为浮点数尾数的 n 位原码小数相除时，只要在被除数 X 的低位添加 n 个0即可。即将 $X=0.x_{n-1}x_{n-2}\cdots x_1x_0$ 变成 $X=0.x_{n-1}x_{n-2}\cdots x_1x_000\cdots00$，显然，扩展为 $2n$ 位后，R 寄存器中为被除数 X，Q 寄存器中为全0。这种情况下，如果|被除数|≥|除数|，则商的数值部分将进到小数点前面的整数位上而发生溢出。

(3) 对于一个 $2n$ 位的数与一个 n 位的数相除的情况，则无须对被除数 X 进行扩展，这种情况下，商的位数可能多于 n 位，因此，有可能发生溢出。采用这种方式的机器，其除法指令给出的被除数在两个寄存器或一个双倍字长寄存器中，这种方式通常称为双精度除法。

综合上述几种情况，可把定点正整数和定点正小数归结在统一的假设下，并将其统称为无符号数的除法。即假设除法运算时，被除数 X 为 $2n$ 位，$X=x_{2n-1}x_{2n-2}\cdots x_n\cdots x_1x_0$，除数 Y 和商 Q 都为 n 位，$Y=y_{n-1}y_{n-2}\cdots y_1y_0$，$Q=q_{n-1}q_{n-2}\cdots q_1q_0$。本书后面对无符号数除法和原码定点小数除法的算法描述都基于这个假设。

参考手工除法过程，得到计算机中两个无符号数除法的运算步骤和算法要点如下。

① 操作数预置：在确认被除数和除数都不为0后，将被除数(必要时进行0扩展)置于余数寄存器 R 和余数/商寄存器 Q 中，除数置于除数寄存器 Y 中。

② 做减法试商：根据 $R-Y$ 得到的结果的符号来判断两数的大小。若结果为正，则上商1，若结果为负，则上商0。

③ 上商为0时恢复余数：把减掉的除数再加回来，恢复原来的中间余数。

④ 中间余数左移，以便继续试商：手算除法中，每次试商前，除数右移与中间余数进行比较。在计算机内部进行除法运算时，除数在除数寄存器中不动，因此，需要将中间余数左移来与除数相减进行比较。左移时，中间余数和商一起进行，余数/商寄存器 Q 的最低位空出，以备上商。

上述给出的算法要点③中，采用了“上商为0时恢复余数”的方式，所以，把上述这种方法称为“恢复余数法”。也可以不这样做，而是在下一步运算时把当前多减的除数补回来。这种方法称为“不恢复余数法”，又称“加减交替法”。根据余数恢复方式的不同，有“恢复余数除法”和“不恢复余数除法”两种。

1. 恢复余数除法

假定除法运算时，被除数 X 为 $2n$ 位，除数 Y 和商 Q 都为 n 位，X、Y 和 Q 各自表示为：$X=x_{2n-1}x_{2n-2}\cdots x_n\cdots x_1x_0$，$Y=y_{n-1}y_{n-2}\cdots y_1y_0$，$Q=q_{n-1}q_{n-2}\cdots q_1q_0$ 则恢复余数除法的算法步骤如下。

第 1 步：$R_1=X-Y$，若 $R_1<0$，则上商 $q_n=0$，同时恢复余数，即 $R_1=R_1+Y$；若 $R_1\geqslant0$，则上商 $q_n=1$。

这里求得的商 q_n 是商的第 n 位数值，不是真正商中的数位。显然，若 $q_n=1$，则商将会有 $n+1$ 位数。这对不同类型数据来说，情况不一样。

(1) 对于无符号整数除法来说，如果被除数为 $2n$ 位，则商有可能会超出 n 位无符号整数范围，所以，此时，若 $q_n=1$，则发生溢出。

(2) 对于原码定点小数除法来说，若 $q_n=1$，则相除结果的数值从小数部分溢出到了整数部分，按道理两个定点小数相除，结果也应是定点小数，故应当作溢出处理。但浮点数尾数溢出时，可通过右规来消除，最终只要阶码不溢出，结果仍然正确。所以，这种情况下，保留最高位的商 $q_n=1$，继续执行下去。

第 2 步：若已求得第 i 次的中间余数为 R_i，则第 $i+1$ 次的中间余数为 $R_{i+1}=2R_i-Y$。若 $R_{i+1}<0$，则上商 $q_{n-i}=0$，同时恢复余数 $R_{i+1}=R_{i+1}+Y$；若 $R_{i+1}\geqslant0$，则上商 $q_{n-i}=1$。

第 3 步：循环执行第 2 步 n 次，直到求出所有 n 位商“$q_{n-1}\sim q_0$”为止。

最终商在 Q 寄存器中，余数在 R 寄存器中。图 3.27 是上述恢复余数除法算法的流程图。

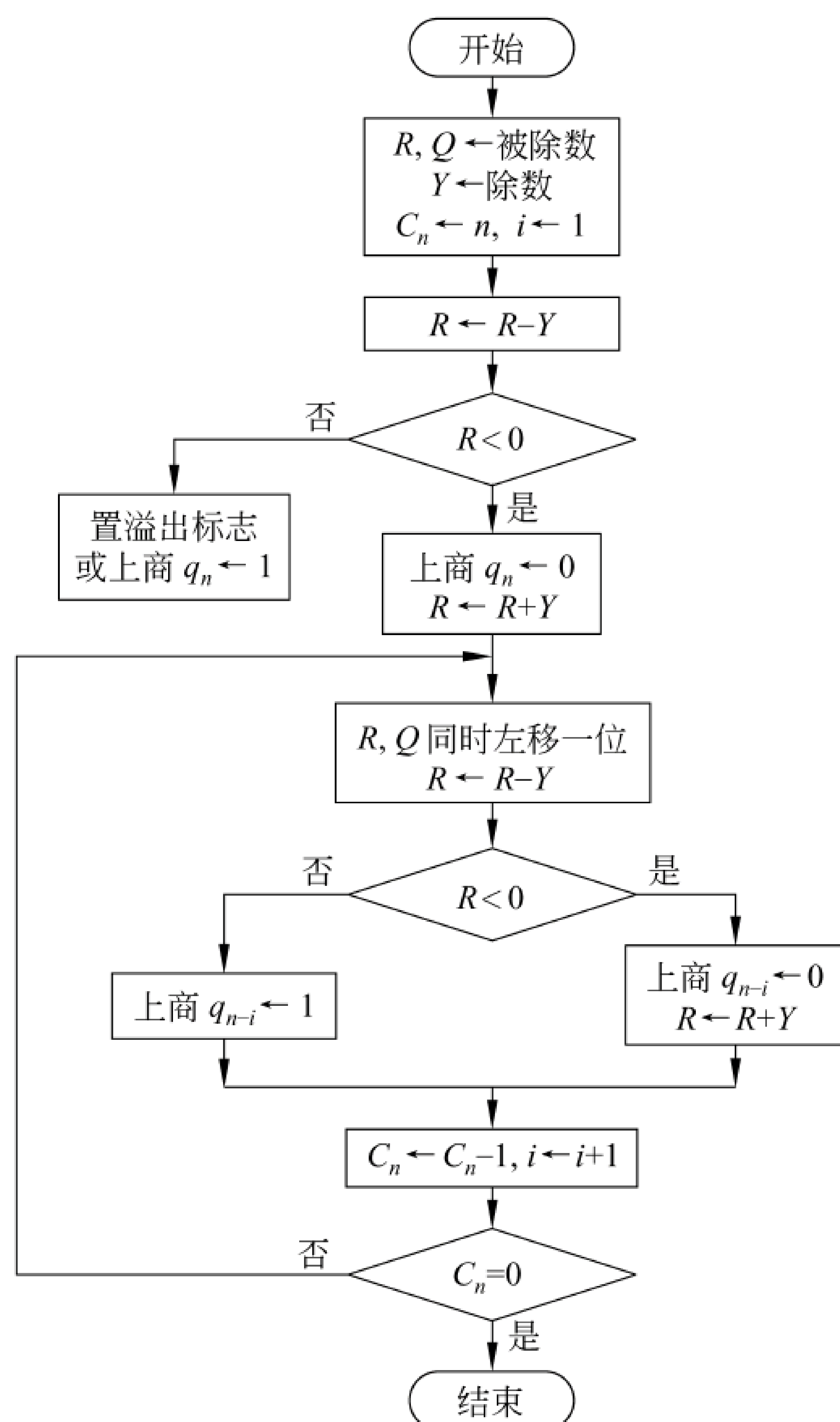


图 3.27　恢复余数除法的运算流程图

例 3.11 已知$[X]_{原}=0.1011$，$[Y]_{原}=1.1101$，用恢复余数法计算$[X/Y]_{原}$。

解：分符号位和数值位两部分进行。商的符号位：$0\oplus1=1$。

商的数值位采用恢复余数法。减法操作用补码加法实现，是否够减通过中间余数的符号来判断，所以中间余数要加一位符号位。因此，需先计算出$[|X|]_{补}=0.1011$，$[|Y|]_{补}=0.1101$，$[-|Y|]_{补}=1.0011$。

因为是原码定点小数，所以在低位扩展0。虽然实际参加运算的数据是$[|X|]_{补}$和$[|Y|]_{补}$，但为简单起见，说明时分别标识为X和Y。运算过程如下。

余数寄存器 R	余数/商寄存器 Q	说明
0 1 0 1 1	0 0 0 0 □	开始 $R_0=X$
+1 0 0 1 1		$R_1=X-Y$
1 1 1 1 0	0 0 0 0 0	$R_1<0$，则 $q_4=0$
+0 1 1 0 1		恢复余数：$R_1=R_1+Y$
0 1 0 1 1		得 R_1
1 0 1 1 0	0 0 0 0 □	$2R_1$(R和Q同时左移，空出一位商)
+1 0 0 1 1		$R_2=2R_1-Y$
0 1 0 0 1	0 0 0 0 1	$R_2>0$，则 $q_3=1$
1 0 0 1 0	0 0 0 1 □	$2R_2$(R和Q同时左移，空出一位商)
+1 0 0 1 1		$R_3=2R_2-Y$
0 0 1 0 1	0 0 0 1 1	$R_3>0$，则 $q_2=1$
0 1 0 1 0	0 0 1 1 □	$2R_3$(R和Q同时左移，空出一位商)
+1 0 0 1 1		$R_3=2R_2-Y$
1 1 1 0 1	0 0 1 1 0	$R_4<0$，则 $q_1=0$
+0 1 1 0 1		恢复余数：$R_4=R_4+Y$
0 1 0 1 0	0 0 1 1 0	得 R_4
1 0 1 0 0	0 1 1 0 □	$2R_4$(R和Q同时左移，空出一位商)
+1 0 0 1 1		$R_5=2R_4-Y$
0 0 1 1 1	0 1 1 0 1	$R_5>0$，则 $q_0=1$

商的最高位为0，说明没有溢出，商的数值部分为1101。

所以，$[X/Y]_{原}=1.1101$（最高位为符号位），余数为0.0111×2^{-4}。

例 3.12 假定X和Y是两个4位无符号数，$X=1101$，$Y=1011$，用恢复余数法计算X/Y。

解：首先对被除数进行0扩展为$X=0000\ 1101$。减法操作用补码加法实现，而且是否够减通过中间余数的符号来判断，所以中间余数要加一位符号位。因此，需先计算出$[|X|]_{补}=0\ 0000\ 1101$，$[|Y|]_{补}=0\ 1011$，$[-|Y|]_{补}=1\ 0101$，虽然实际参加运算的数据是$[|X|]_{补}$和$[|Y|]_{补}$，但为简单起见，分别标识为X和Y，运算过程如下。

余数寄存器 R	余数/商寄存器 Q	说　　明
00000	1101□	开始 $R_0=X$
+10101		$R_1=X-Y$
10101	1101	$R_1<0$，则 $q_4=0$
+01011		恢复余数：$R_1=R_1+Y$
00000		得 R_1
00001	1010□	$2R_1$(R 和 Q 同时左移，空出一位商)
+10101		$R_2=2R_1-Y$
10110	10100	$R_2<0$，则 $q_3=0$
+01011		恢复余数：$R_2=R_2+Y$
00001		得 R_2
00011	0100□	$2R_2$(R 和 Q 同时左移，空出一位商)
+10101		$R_3=2R_2-Y$
11000	01000	$R_3<0$，则 $q_2=0$
+01011		恢复余数：$R_3=R_3+Y$
00011		得 R_3
00110	1000□	$2R_3$(R 和 Q 同时左移，空出一位商)
+10101		$R_3=2R_2-Y$
11011	10000	$R_4<0$，则 $q_1=0$
+01011		恢复余数：$R_4=R_4+Y$
00110	10000	得 R_4
01101	0000□	$2R_4$(R 和 Q 同时左移，空出一位商)
+10101		$R_5=2R_4-Y$
00010	00001	$R_5>0$，则 $q_0=1$

商的最高位为 0，说明没有溢出（理论上两个 4 位无符号数相除一定不会溢出）。

结果得到 4 位无符号商为 0001，余数为 0010，将各数代入公式“除数×商＋余数＝被除数”进行验证，得 1011×0001＋0010＝1101，结果正确。

2. 不恢复余数除法

在恢复余数除法运算中，当中间余数与除数相减结果为负时，要多做一次 $+Y$ 操作，降低了算法执行速度，又使控制线路变得复杂。在计算机中很少采用恢复余数除法，而普遍采用不恢复余数除法。

在恢复余数除法中，第 i 次余数为 $R_i=2R_{i-1}-Y$。根据下次中间余数的计算方法，有以下两种不同的情况：

若 $R_i \geqslant 0$ 时，则上商 1，不需恢复余数，直接左移一位后试商，即 $R_{i+1}=2R_i-Y$。

若 $R_i<0$ 时，则上商 0，需恢复余数后左移一位再试商，即 $R_{i+1}=2(R_i+Y)-Y=2R_i+Y$。

从上述结果可知，当第 i 次中间余数为负时，可以跳过恢复余数这一步，直接求第 $i+1$ 次中间余数。这种算法称为不恢复余数法。从上述推导可以发现，不恢复余数法的算法要点就是 6 个字：“正、1、减，负、0、加”。其含义就是：若中间余数为正数，则上商为 1，下次做减法；若中间余数为负数，则上商为 0，下次做加法。这样运算中每次循环内的步骤都是规整的，差别仅在做加法还是减法，所以，这种方法也称为“加减交替法”。采用这种方法有一点要注意，即如果在最后一步上商为 0，则必须恢复余数，把试商时减掉的除数加回去。

假定被除数 X 为 $2n$ 位，除数 Y 为 n 位。$X=x_{2n-1}x_{2n-2}\cdots x_n\cdots x_1x_0$，$Y=y_{n-1}y_{n-2}\cdots y_1y_0$，用不恢复余数法求解 X/Y 的商和余数的运算流程如图 3.28 所示。

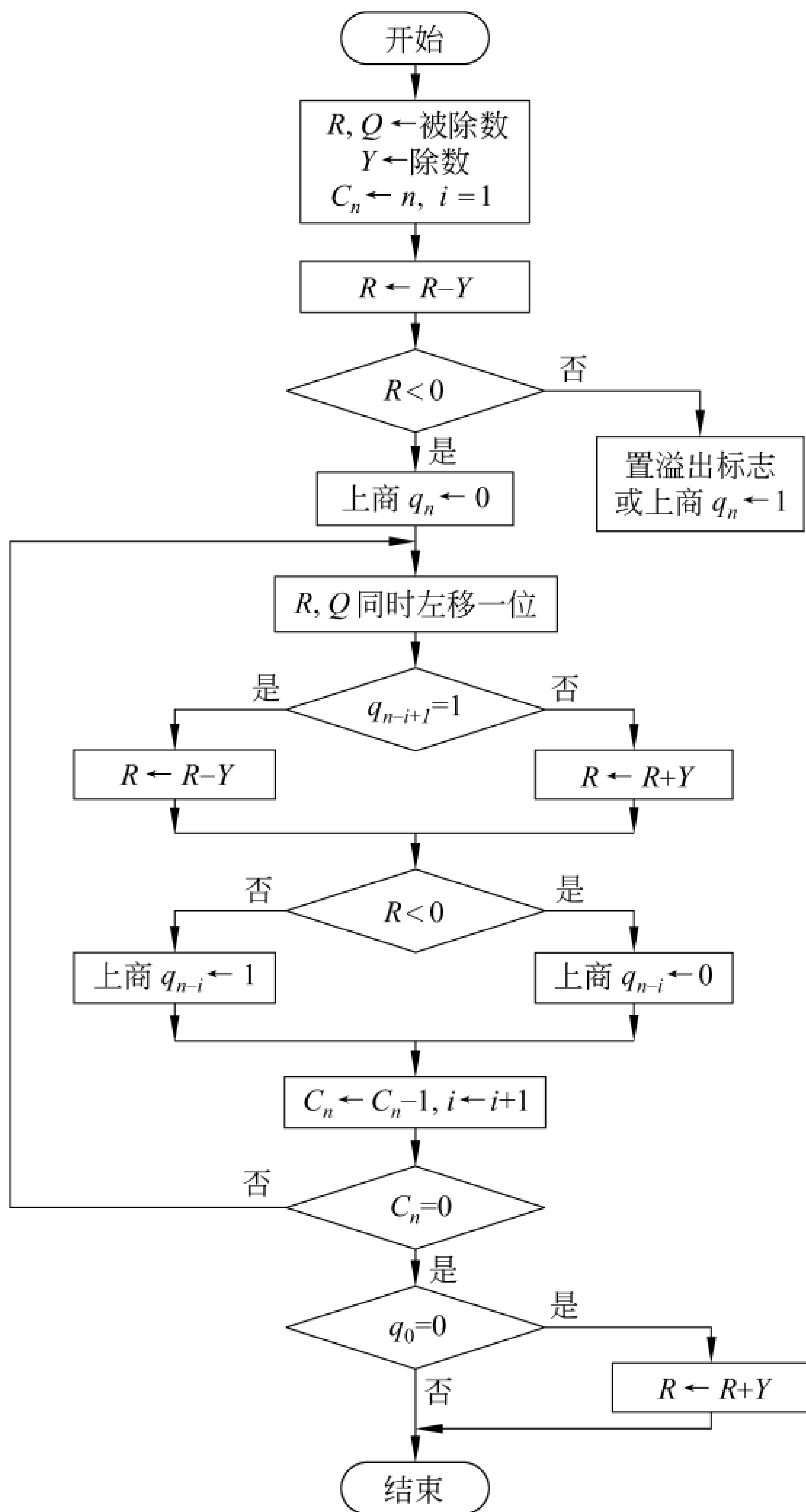


图 3.28　不恢复余数除法运算流程

例 3.13　已知$[X]_{原}=0.1011$，$[Y]_{原}=1.1101$，用不恢复余数法计算$[X/Y]_{原}$。

解：分符号位和数值位两部分进行。商的符号位：$0\oplus1=1$。

商的数值位采用不恢复余数法。减法操作用补码加法实现，是否够减通过中间余数的符号来判断，所以中间余数要加一位符号位。因此，需先计算出 $[|X|]_{补}=0.1011$，$[|Y|]_{补}=0.1101$，$[-|Y|]_{补}=1.0011$。

因为是原码定点小数，所以在低位扩展 0。虽然实际参加运算的数据是 $[|X|]_{补}$ 和 $[|Y|]_{补}$，但为简单起见，说明时分别标识为 X 和 Y。运算过程如下：

余数寄存器 R	余数/商寄存器 Q	说　明
01011	0000□	开始 $R_0=X$
+10011		$R_1=X-Y$
11110	00000	$R_1<0$，则 $q_4=0$，没有溢出
11100	0000□	$2R_1$(R 和 Q 同时左移，空出一位商)
+01101		$R_2=2R_1+Y$
01001	00001	$R_2>0$，则 $q_3=1$
10010	0001□	$2R_2$(R 和 Q 同时左移，空出一位商)
+10011		$R_3=2R_2-Y$
00101	00011	$R_3>0$，则 $q_2=1$
01010	0011□	$2R_3$(R 和 Q 同时左移，空出一位商)
+10011		$R_3=2R_2-Y$
11101	00110	$R_4<0$，则 $q_1=0$
11010	0110□	$2R_4$(R 和 Q 同时左移，空出一位商)
+01101		$R_5=2R_4+Y$
00111	01101	$R_5>0$，则 $q_0=1$

商的最高位为 0，说明没有溢出，商的数值部分为 1101。所以，$[X/Y]_{原}=1.1101$(最高位为符号位)，余数为 0.0111×2^{-4}。

从上述给出的几个除法例子以及有关恢复余数法和不恢复余数法的算法流程中，可以看出，要得到 n 位无符号数的商，需要循环 $n+1$ 次，其中第一次得到的不是真正的商，而是用来判断溢出的。为了节省运算时间，第一次可以不试商而直接左移，这样只要 n 次循环。因为对于两个 n 位定点整数除法来说，其商一定不会超过 n 位，所以不会发生溢出，因而，n 位定点整数除法第一次无须试商来判断溢出。

*3.3.8　补码除法运算

补码作为机器中带符号整数的表示形式，需要计算机能实现定点补码整数的除法运算。与补码加减运算、补码乘法运算一样，补码除法也可以将符号位和数值位合在一起进行运算，而且商符直接在除法运算中产生。对于两个 n 位补码的除法运算，被除数需要进行符号扩展。若被除数为 $2n$ 位，除数为 n 位，则被除数无须扩展。

前面介绍过在进行定点数除法运算前，首先要对被除数和除数的取值、大小等进行相应的判断，以确定除数是否为 0、商是否为 0、是否溢出或为不确定的值 NaN。在确定不会发生上述这些特殊情况的前提下，才会继续执行除法运算。

因为补码除法中被除数、中间余数和除数都是有符号的，所以，不像无符号除法和原码除法那样，可以直接用做减法来判断是否够减，而应该根据被除数(或中间余数)与除数之间符号的异同和差值的正负来确定下次做减法还是加法，再根据加或减运算的结果来判断是否够减。其判断是否够减的规则如下：

(1) 当被除数(或中间余数)与除数同号时，做减法，若得到的新余数的符号与除数符号一致表示够减，否则为不够减。

(2) 当被除数(或中间余数)与除数异号时，做加法，若得到的新余数的符号与除数符号一致表示不够减，否则为够减。

由此，得到表 3.5 中有关补码除法判断是否够减的规则。

表 3.5　补码除法判断是否够减的规则

中间余数 R	除数 Y	新中间余数：$R-Y$		新中间余数：$R+Y$	
		0	1	0	1
0	0	够减	不够减		
0	1			够减	不够减
1	0			不够减	够减
1	1	不够减	够减		

上述判断是否够减的规则是根据新余数和除数之间符号的异同来进行的，也可以根据加减前、后的中间余数的符号变化来判断是否够减。

根据表 3.5 可得到其判断规则如下：

(1) 当被除数(或中间余数)与除数同号时，做减法；异号时，做加法。

(2) 若加减运算后得到的新余数符号与原余数符号一致，即余数符号未改变，表示够减；否则表示不够减。

根据是否立即恢复余数，补码除法也分为恢复余数法和不恢复余数法两种。

1. 补码恢复余数除法

根据补码除法判断是否够减的判断规则，得到如下补码恢复余数除法的算法要点。

(1) 操作数的预置。除数装入除数寄存器 Y，被除数符号扩展后装入余数寄存器 R 和余数/商寄存器 Q。

(2) R 和 Q 同步串行左移一位。

(3) 若 R 与 Y 同号，则做减法，即 $R=R-Y$；否则做加法，即 $R=R+Y$，并按以下规则确定商值 q_0。

① 若 R 和 Q 中的中间余数 $=0$ 或 R 操作前后符号未变，则表示够减，q_0 置 1，转第(4)步。

② 若 R 操作前后符号已变，则表示不够减，q_0 置 0，恢复 R 值后转第(4)步。

(4) 重复第(2)和第(3)步，直到取得 n 位商为止。

(5) 若被除数与除数同号，则 Q 中是真正的商；否则，将 Q 求补。

(6) 余数在 R 中。

从上述算法要点可以看出，在进行置商时，采用了“够减则上商为 1，不够减则上商为 0”的上商方式，因此，最后若商为负值，则需要“各位取反，末位加 1”来得到真正的商。

例 3.14　用 4 位补码恢复余数法计算 7/3 和 −7/3 的值。

解：被除数 7 的 8 位补码表示为 $[X]_{补}=0000\ 0111$，除数 3 的 4 位补码表示为 $[Y]_{补}=0011$，$[-Y]_{补}=1101$。

余数寄存器 R	余数/商寄存器 Q	说 明
0000	0111	开始 R_0=[X]
0000	111□	$2R_0$(R和Q同时左移，空出一位商)
+1101		R与Y同号，减
1101	111□	R符号已变
+0011		恢复(加)
0000	1110	q_0置0，得R_1
0001	110□	$2R_1$(R和Q同时左移，空出一位商)
+1101		R与Y同号，减
1110	110□	R符号已变
+0011		恢复(加)
0001	1100	q_0置0，得R_2
0011	100□	$2R_2$(R和Q同时左移，空出一位商)
+1101		R与Y同号，减
0000	1001	R符号未变，q_0置1，得R_3
0001	001□	$2R_3$(R和Q同时左移，空出一位商)
+1101		R与Y同号，减
1110	001□	R符号已变
+0011		恢复(加)
0001	0010	q_0置0，得R_4

所以，商的补码为0010，余数的补码为0001。即7/3=2，余数为1。

被除数−7的8位补码为$[X]_{补}$=1111 1001，除数3的4位补码为$[Y]_{补}$=0011，$[-Y]_{补}$=1101。

余数寄存器 R	余数/商寄存器 Q	说 明
1111	1001	开始 R_0=[X]
1111	001□	$2R_0$(R和Q同时左移，空出一位商)
+0011		R与Y异号，加
0010	001□	R符号已变
+1101		恢复(减)
1111	0010	q_0置0，得R_1
1110	010□	$2R_1$(R和Q同时左移，空出一位商)
+0011		R与Y异号，加
0001	010□	R符号已变
+1101		恢复(减)
1110	0100	q_0置0，得R_2
1100	100□	$2R_2$(R和Q同时左移，空出一位商)
+0011		R与Y异号，加
1111	1001	R符号未变，q_0置1，得R_3
1111	001□	$2R_3$(R和Q同时左移，空出一位商)
+0011		R与Y异号，加
0010	001□	R符号已变
+1101		恢复(减)
1111	0010	q_0置0，得R_4

被除数与除数异号，故商取补为1110，余数为1111，即−7/3=−2，余数为−1。

2. 补码不恢复余数除法

根据补码除法判断是否够减的规则，得到如下补码不恢复余数除法的算法要点。

(1) 操作数的预置。除数装入除数寄存器 Y，被除数符号扩展后装入余数寄存器 R 和余数/商寄存器 Q。

(2) 根据以下规则求第一位商 q_n。

若被除数 X 与 Y 同号，则做减法，即 $R_1=X-Y$；否则做加法，即 $R_1=X+Y$，按以下规则确定商值 q_n。

① 若新的中间余数 R_1 与 Y 同号，则 q_n 置 1，转下一步。

② 若新的中间余数 R_1 与 Y 异号，则 q_n 置 0，转下一步。

q_n 用来判断是否溢出，而不是真正的商。以下情况下会发生溢出：X 与 Y 同号且上商 $q_n=1$，或者，X 与 Y 异号且上商 $q_n=0$。

(3) 对于 $i=1$ 到 n，按以下规则求出 n 位商。

① 若 R_i 与 Y 同号，则 q_{n-i} 置 1，$R_{i+1}=2R_i-[Y]_{补}$，$i=i+1$。

② 若 R_i 与 Y 异号，则 q_{n-i} 置 0，$R_{i+1}=2R_i+[Y]_{补}$，$i=i+1$。

(4) 商的修正：最后一次 Q 寄存器左移一位，将最高位 q_n 移出，在最低位置上商 q_0。若被除数与除数同号，Q 中就是真正的商；否则，将 Q 中的商的末位加 1。

(5) 余数的修正：若余数符号同被除数符号，则不需修正，余数在 R 中；否则，按下列规则进行修正：当被除数和除数符号相同时，最后余数加除数；否则，最后余数减除数。

与无符号数的不恢复余数法一样，补码不恢复余数法也有一个 6 字口诀“同、1、减，异、0、加”。所以，其运算过程也是呈加减交替的方式，因此也称为“加减交替法”。

例 3.15 已知 $X=-9$，$Y=2$，要求用补码除法计算 $[X/Y]_{补}$。

解：$[X]_{补}=1\ 0111$，$[Y]_{补}=0\ 0010$，计算过程如下：

先对被除数进行符号扩展，$[X]_{补}=11111\ 10111$，$[-Y]_{补}=1\ 1110$。

余数寄存器 R	余数/商寄存器 Q	说　明
1 1 1 1 1	1 0 1 1 1	开始 $R_0=[X]$
+0 0 0 1 0		$R_1=[X]+[Y]$
0 0 0 0 1	1 0 1 1 1	R_1 与 $[Y]$ 同号，则 $q_5=1$
0 0 0 1 1	0 1 1 1 | 1	$2R_1$(R 和 Q 同时左移，空出一位上商 1)
+1 1 1 1 0		$R_2=2R_1+[-Y]$
0 0 0 0 1	0 1 1 1 | 1	R_2 与 $[Y]$ 同号，则 $q_4=1$，
0 0 0 1 0	1 1 1 | 1 1	$2R_2$(R 和 Q 同时左移，空出一位上商 1)
+1 1 1 1 0		$R_3=2R_2+[-Y]$
0 0 0 0 0	1 1 1 | 1 1	R_3 与 $[Y]$ 同号，则 $q_3=1$
0 0 0 0 1	1 1 | 1 1 1	$2R_3$(R 和 Q 同时左移，空出一位上商 1)
+1 1 1 1 0		$R_4=2R_3+[-Y]$
1 1 1 1 1	1 1 | 1 1 1	R_4 与 $[Y]$ 异号，则 $q_2=0$
1 1 1 1 1	1 | 1 1 1 0	$2R_4$(R 和 Q 同时左移，空出一位上商 0)
+0 0 0 1 0		$R_5=2R_4+[Y]$
0 0 0 0 1	1 | 1 1 1 0	R_5 与 $[Y]$ 同号，则 $q_1=1$，
0 0 0 1 1	| 1 1 1 0 1	$2R_5$(R 和 Q 同时左移，空出一位上商 1)
+1 1 1 1 0		$R_6=2R_5+[-Y]$
0 0 0 0 1	| 1 1 0 1 1	R_6 与 $[Y]$ 同号，则 $q_0=1$，Q 左移，空出一位上商 1
+1 1 1 1 0	+　　　1	商为负数，末位加 1；减除数以修正余数
1 1 1 1 1	1 1 1 0 0	

所以，$[X/Y]_{补}=11100$，余数为 11111。

即 $X/Y=-0100B=-4$，余数为 $-0001B=-1$。

将各数代入公式"除数×商+余数=被除数"进行验证，得 $2\times(-4)+(-1)=-9$。

*3.3.9 阵列除法器

与乘法类似，除法运算也可以采用专门的阵列除法器。如图 3.29 是一个阵列除法器示意图。其中图 3.29(a)是一个组合逻辑单元，称为可控加减单元(CAS)。该单元中有一个异或门和一个全加器，有 4 个输入端和 4 个输出端。x 和 y 为一位操作数，ci 为低位传送来的进位或借位信号，co 为本单元送往高位的进位/借位信号，s 为本单元的结果，即本位的和或差。p 为控制信号，一方面决定本单元 s 的值，另一方面也串行传送给低位。当 $p=0$ 时，CAS 作为全加器单元，当 $p=1$ 时，输入 y 被取反，CAS 作为全减器单元。

该单元中的异或门实现 p 和 y 的异或，其输出再作为全加器的输入，所以 CAS 电路的逻辑表达式为：

$$s = x \oplus (p \oplus y) \oplus \mathrm{ci}$$
$$\mathrm{co} = (x + \mathrm{ci})(p \oplus y) + x\mathrm{ci}$$

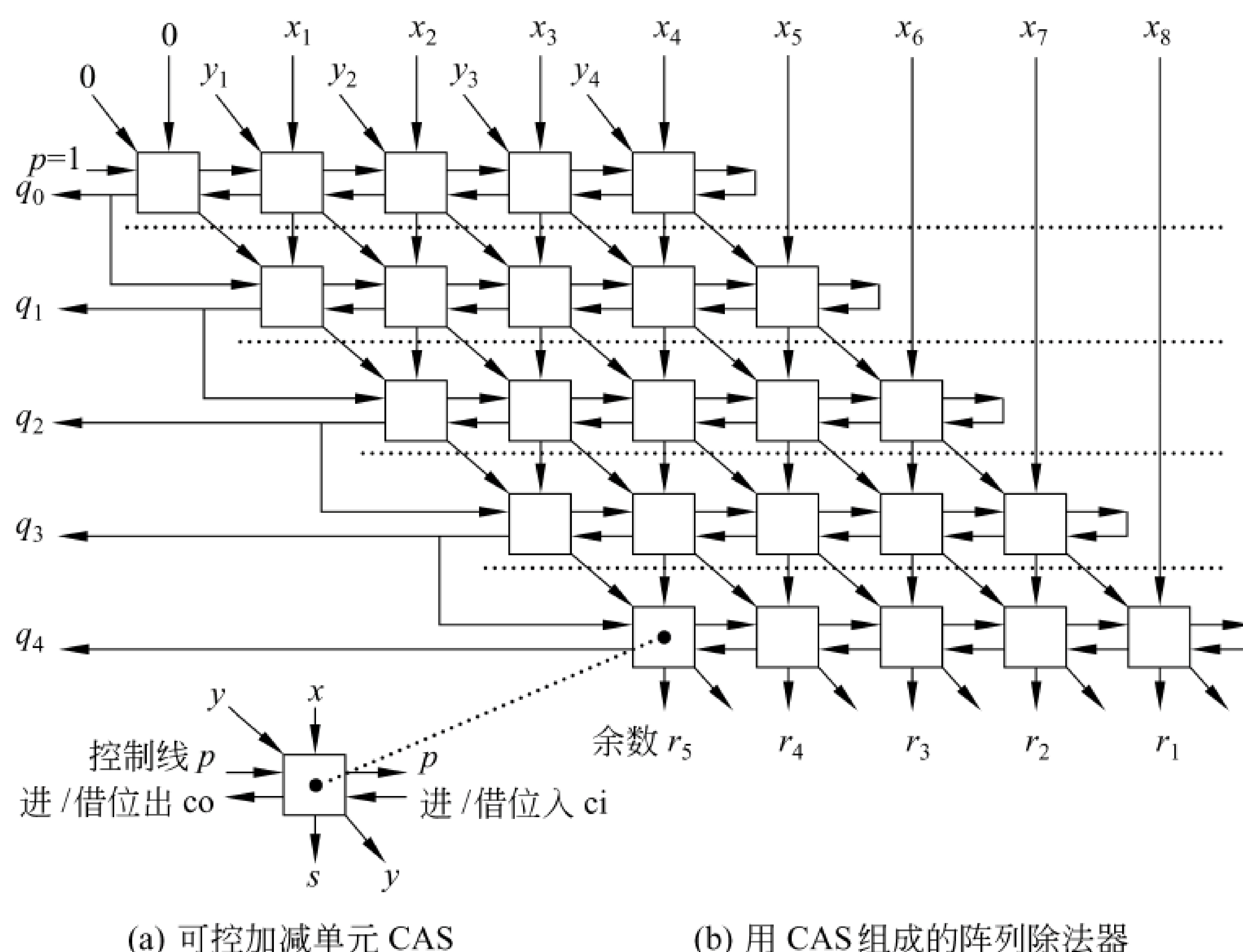
由此可知：

当 $p=0$ 时，　$s=x\oplus y\oplus \mathrm{ci}$，　$\mathrm{co}=xy+y\mathrm{ci}+x\mathrm{ci}$。

当 $p=1$ 时，　$s=x\oplus \overline{y}\oplus \mathrm{ci}$，　$\mathrm{co}=x\,\overline{y}+\overline{y}\mathrm{ci}+x\mathrm{ci}$。

因此，当 $p=1$ 时，各 CAS 电路实现了 y 取反的功能，若将 $p=1$ 接到最低位 CAS 的 ci 端，则可实现末位加 1 的功能。根据 3.3.1 节介绍的补码加减运算可知，整个电路实现了补码减法的功能。

图 3.29(b)是一个用于对两个正数按不恢复余数法进行相除的阵列除法器。在该阵



(a) 可控加减单元 CAS　　(b) 用 CAS 组成的阵列除法器

图 3.29　不恢复余数法的阵列除法器

列除法器中，每行最高位单元的 p 和最低位的输入进位 ci 连在一起。因为无符号数的不恢复余数法第一次总是做减法，所以，第一行的控制线 p 总是固定为 1。这样，最低位的进位输入也为 1，实现了末位加 1，同时第一行每个 CAS 中对除数 y 取反，因此，在第一行实现了减法。不恢复余数法的要诀是"正、1、减，负、0、加"，也即中间余数的符号决定了对应位的商的值以及下次是做加法还是减法。阵列除法器中，中间余数的符号就是每一行最高位 CAS 的进位输出。因此，每一行最高位进位输出要同时接到对应位的商 q_i 和下一行控制位 p 上。

3.4 浮点数运算

从第 3.1 节介绍的有关高级语言和机器指令涉及到的运算来看，浮点运算主要包括符点数的加、减、乘、除运算。一般有单精度浮点数和双精度浮点数运算，有些机器还支持 80 位或 128 位扩展浮点数运算。

3.4.1 浮点数加减运算

先看一个十进制数加法运算的例子：$0.123\times10^5+0.456\times10^2$。显然，不可以把 0.123 和 0.456 直接相加，必须把指数调整为相等后才可实现两数相加，其计算过程如下。

$$\begin{aligned}0.123\times10^5+0.456\times10^2&=0.123\times10^5+0.000456\times10^5\\&=(0.123+0.000456)\times10^5=0.123456\times10^5\end{aligned}$$

从上面的例子不难理解实现浮点数加减法的运算规则。

设两个规格化浮点数 X 和 Y 表示为 $X=M_x\times2^{Ex}$，$Y=M_y\times2^{Ey}$，M_x、M_y 分别是浮点数 X 和 Y 的尾数，E_x、E_y 分别是浮点数 X 和 Y 的阶码，不失一般性，设 $E_x\leqslant E_y$，那么

$$X+Y=(M_x\times2^{Ex-Ey}+M_y)\times2^{Ey}$$

$$X-Y=(M_x\times2^{Ex-Ey}-M_y)\times2^{Ey}$$

计算机中实现上述计算过程需要经过对阶、尾数加减、规格化和舍入 4 个步骤，此外，还必须考虑溢出判断和溢出处理问题。假定在下面的讨论中 $X\pm Y$ 未经规格化的结果表示为 $M_b\times2^{Eb}$。

1. 对阶

对阶的目的是使 X 和 Y 的阶码相等，以使尾数可以相加减。对阶的原则是：小阶向大阶看齐，阶小的那个数的尾数右移，右移的位数等于两个阶码差的绝对值。

假设 $\Delta E=E_x-E_y$，则对阶操作可以表示如下：

若 $\Delta E\leqslant0$，则 $E_b\leftarrow E_y$，　$E_x\leftarrow E_y$，　$M_x\leftarrow M_x\times2^{Ex-Ey}$。

若 $\Delta E>0$，则 $E_b\leftarrow E_x$，　$E_y\leftarrow E_x$，　$M_y\leftarrow M_y\times2^{Ey-Ex}$。

大多数机器采用 IEEE 754 标准来表示浮点数，因此，对阶时需要进行移码减法运算，并且尾数右移时按原码小数方式右移，符号位不参加移位，数值位要将隐含的一位"1"右移到小数部分，空出位补 0。为了保证运算的精度，尾数右移时，低位移出的位不要丢掉，应保留并参加尾数部分的运算。

根据3.3.3节介绍的有关移码加减运算规则，可知：

$$[\Delta E]_{补}=[E_x-E_y]_{补}=[E_x]_{移}+[-[E_y]_{移}]_{补}\quad(\text{mod } 2^n)$$

因此，只要先对$[E_y]_{移}$求补，再与$[E_x]_{移}$相加，就可以计算$[\Delta E]_{补}$。对$[E_y]_{移}$求补时采用"各位取反，末位加1"即可。然后根据$[\Delta E]_{补}$的符号，可以判断出$\Delta E>0$还是$\Delta E\leqslant 0$。

2. 尾数加减

对阶后两个浮点数的阶码相等，此时，可以进行对阶后的尾数相加减。因为IEEE 754采用定点原码小数表示尾数，所以，尾数加减实际上是定点原码小数的加减运算，可根据3.3.2节介绍的定点原码小数加减运算进行。因为，IEEE 754浮点数尾数中有一个隐藏位，所以，在进行尾数加减时，必须把隐藏位还原到尾数部分。此外，对阶过程中在尾数右移时保留的附加位也要参加运算。因此，在用定点原码小数进行尾数加减运算时，在操作数的高位部分和低位部分都需要进行相应的调整。

进行加减运算后的尾数不一定是规格化的，因此，浮点数的加、减运算需要进一步进行规格化处理。

3. 尾数规格化

IEEE 754的规格化尾数形式为$\pm 1.xx\cdots x$。在进行尾数相加减后可能会得到各种形式的结果，例如，

$$1.xx\cdots x+1.xx\cdots x=\pm 1x.xx\cdots x$$

$$1.xx\cdots x-1.xx\cdots x=\pm 0.00\cdots 01x\cdots x$$

(1) 对于上述结果为$\pm 1x.xx\cdots x$的情况，需要进行右规：尾数右移一位，阶码加1。右规操作可以表示为$M_b\leftarrow M_b\times 2^{-1}$，$E_b\leftarrow E_b+1$。尾数右规时，最高位"1"被移到小数点前一位作为隐藏位，最后一位移出时，要考虑舍入。阶码加1时，直接在末位加1。

(2) 对于上述结果为$\pm 0.00\cdots 01x\cdots x$的情况，需要进行左规：数值位逐次左移，阶码逐次减1，直到将第一位"1"移到小数点左边。假定k为结果中"±"和最左边第一个1之间连续0的个数，则左规操作可以表示为$M_b\leftarrow M_b\times 2^k$，$E_b\leftarrow E_b-k$。

尾数左规时数值部分最左k个0被移出，因此，相对来说，小数点右移了k位。因为进行尾数相加时，默认小数点位置在第一个数值位(即隐藏位)之后，所以小数点右移k位后被移到了第一位1后面，这个1就是隐藏位。执行$E_b\leftarrow E_b-k$时，每次都在末位减1，一共减k次。

4. 尾数的舍入处理

在对阶和尾数右规时，可能会对尾数进行右移，为保证运算精度，一般将低位移出的位保留下来，参加中间过程的运算，最后再将运算结果进行舍入，还原表示成IEEE 754格式。这里要解决两个问题。

(1) 保留多少附加位才能保证运算的精度？

(2) 最终如何对保留的附加位进行舍入？

对于第(1)个问题，可能无法给出一个准确的答案。但是不管怎么说，保留附加位一定得到比不保留附加位更高的精度。IEEE 754标准规定，所有浮点数运算的中间结果右边都

必须至少额外保留两位附加位。这两位附加位中，紧跟在浮点数尾数右边那一位为保护位或警戒位(guard)，用以保护尾数右移的位，紧跟在保护位右边的是舍入位(round)，左规时可以根据其值进行舍入。在 IEEE 754 中，为了更进一步提高计算的精度，在保护位和舍入位后面还引入了额外的一个数位，称为粘位(sticky)，只要舍入位的右边有任何非 0 数字，粘位就被置 1；否则，粘位被置为 0。

对于第(2)个问题，IEEE 754 提供了以下可选的 4 种模式。

(1) 就近舍入。舍入为最近可表示的数。当运算结果是两个可表示数的非中间值时，实际上是“0 舍 1 入”方式；当运算结果正好在两个可表示数中间时，根据“就近舍入”的原则就无法操作了。IEEE 754 规定这种情况下，结果强迫为偶数。即：若结果的 LSB 为 1(即奇数)时，则末位加 1；若 LSB 为 0(偶数)时，则直接截取。这样，就保证了结果的 LSB 总是 0(即偶数)。使用粘位可以减少运算结果正好在两个可表示数中间的情况。不失一般性，用一个十进制数计算的例子来说明这样做的好处。

假设要求计算 $1.24\times10^4+5.09\times10^1$(假定科学记数法的精度保留两位小数)，若只使用保护位和舍入位而不使用粘位，则结果为 $1.2400\times10^4+0.0050\times10^4=1.2450\times10^4$。这个结果位于两个相邻可表示数 1.24×10^4 和 1.25×10^4 的中间，采用就近舍入到偶数，则结果应该为 1.24×10^4；若同时使用保护位、舍入位和粘位，则结果为 $1.24000\times10^4+0.00509\times10^4=1.24509\times10^4$。这个结果就不在 1.24×10^4 和 1.25×10^4 的中间，而更接近于 1.25×10^4，采用就近舍入方式，结果应该为 1.25×10^4。显然，后者更精确。

(2) 朝$+\infty$方向舍入。总是取右边最近可表示数，也称为正向舍入或朝上舍入。

(3) 朝$-\infty$方向舍入。总是取左边最近可表示数，也称为负向舍入或朝下舍入。

(4) 朝 0 方向舍入。直接截取所需位数，丢弃后面所有位，也称为截取、截断或恒舍法。这种舍入处理最简单。对正数或负数来说，都是取更靠近原点的那个可表示数，是一种趋向原点的舍入，因此又称为趋向零舍入。

5. 阶码溢出判断

在进行尾数规格化和尾数舍入时，可能会对结果的阶码执行加、减运算。因此，必须考虑结果的阶码溢出问题。若一个正阶码超过了最大允许值(127 或 1023)，则发生“阶码上溢”，机器产生“阶码上溢”异常，也有的机器把结果置为“$+\infty$”(数符为 0 时)或“$-\infty$”(数符为 1 时)后，继续执行下去，而不产生“溢出”异常。若一个负阶码超过了最小允许值(-126 或-1022)，则发生“阶码下溢”，此时，一般把结果置为“$+0$”(数符为 0 时)或“-0”(数符为 1 时)，也有的机器引起“阶码下溢”异常。

溢出判断实际上是在上述尾数规格化和尾数舍入过程中进行的，只要涉及到阶码求和/差就可以在阶码运算部件中直接用溢出判断电路来实现。在上述运算过程中，涉及到阶码求和/差的情况有以下几处。

(1) 右规和尾数舍入。一个数值很大的尾数舍入时，可能因为末尾加 1 而发生尾数溢出，此时，可以通过右规来调整尾数和阶码。右规时阶码加 1，导致阶码增大，因此需要判断是否发生了“阶码上溢”。只有当调整前的阶码为 11111110，加 1 后，才会变成 11111111 而发生上溢；如果右规前阶码已经是 11111111，则右规时阶码加 1 后变为 00000000，会造成判

断出错。所以，右规前应先判断阶码是否为全1，若是，则不需右规，直接置结果为阶码上溢；否则，阶码加1，然后判断阶码是否为全1来确定是否阶码上溢。

(2) 左规。左规时数值位逐次左移，阶码逐次减1，所以左规使阶码减小，故需判断是否发生“阶码下溢”。其判断规则与阶码上溢类似，首先判断阶码是否为全0，若是，则直接置阶码下溢；否则，阶码减1，然后判断阶码是否为全0来确定是否阶码下溢。

从浮点数加、减运算过程可以看出，浮点数的溢出并不以尾数溢出来判断，尾数溢出可以通过右规操作得到纠正。运算结果是否溢出主要看结果的阶码是否发生了上溢，因此是由阶码上溢来判断的。

综上所述，规格化浮点数加减运算流程如图3.30所示。

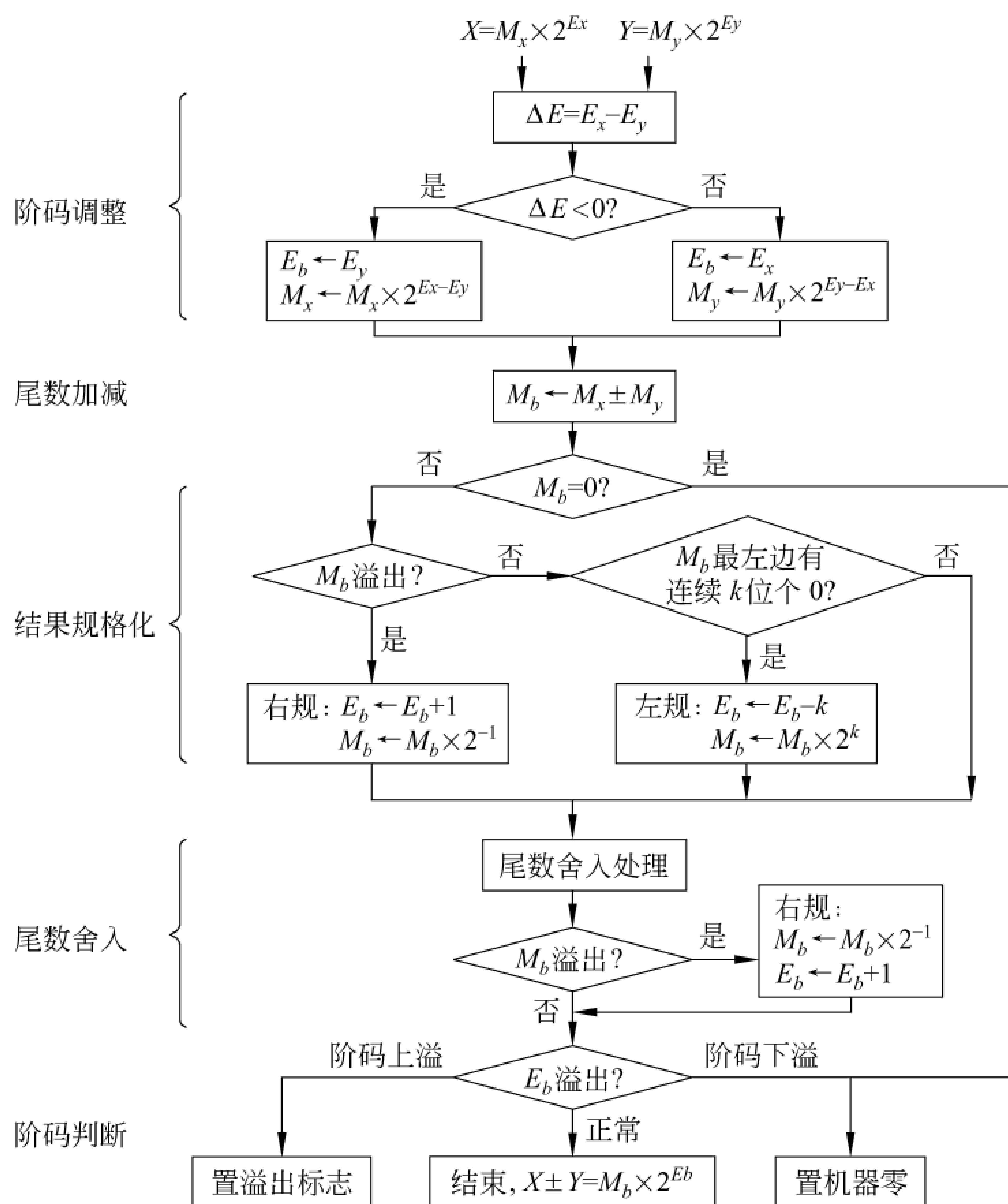


图3.30 浮点数加减法运算流程图

例3.16 用IEEE 754单精度浮点数加减运算计算0.5+(−0.4375)=?

解：$x=0.5=0.100\cdots0\text{B}=(1.00\cdots0)_2\times2^{-1}$。

$y=-0.4375=-0.01110\cdots0\text{B}=(-1.110\cdots0)_2\times 2^{-2}$。

用IEEE 754标准单精度格式表示为：

$$[x]_{浮}=0\ 01111110\ 00\cdots0 \quad [y]_{浮}=1\ 01111101\ 110\cdots0$$

所以，$[E_x]_移=01111110$，$M_x=0\ (1).0\cdots0$，$[E_y]_移=01111101$，$M_y=1(1).110\cdots0$。

尾数 M_x 和 M_y 中小数点前面有两位，第一位为数符，第二位加了括号，是隐藏位“1”。

以下是计算机中进行浮点数加减运算的过程(假定保留两位附加位)。

(1) 对阶

$[\Delta E]_补=[E_x]_移+[-[E_y]_移]_补(\bmod 2^n)=0111\ 1110+1000\ 0011=0000\ 0001$。

因为 $\Delta E=1$，所以需要对 y 进行对阶。即 y 的尾数 M_y 右移一位，符号不变，数值高位补 0，隐藏位右移到小数点后面，最后移出的位保留两位附加位。结果为 $[E_y]_移=[E_x]_移=01111110$，$M_y=10.(1)110\cdots0\mathbf{00}$。

(2) 尾数相加

$M_b=M_x+M_y=01.0000\cdots0\mathbf{00}+10.1110\cdots0\mathbf{00}$(小数点在隐藏位后，最后为附加位)。根据 3.3.2 节介绍的原码加减运算，得 $01.0000\cdots0\mathbf{00}+10.1110\cdots0\mathbf{00}=00.00100\cdots0\mathbf{00}$。上式尾数中最左边第一位是符号位，其余都是数值部分，尾数最后两位是附加位。

(3) 规格化

所得尾数中数值部分高位有 3 个连续的 0，因此需要左规操作。即将尾数左移 3 位，并将阶码减 3。尾数左移时数值部分最左 3 个 0 被移出，小数点右移 3 位后，移到了第一位 1 后面。这个 1 就是隐藏位。因此，得 $M_b=0\ (1).00\cdots0\mathbf{00}000$。

阶码 $E_b=E_b-3=(((01111110-00000001)-00000001)-00000001)=0111\ 1011$。

在计算机中，每次减 1 可通过加$[-1]_补$，即$+11111111$ 来实现。

(4) 舍入

把结果的尾数 M_b 中最后两位舍入掉，从本例来看，不管采用什么舍入法，结果都一样，都是把最后两个 0 去掉，得 $M_b=0\ (1).00\cdots0\mathbf{00}0$。

(5) 溢出判断

在上述阶码计算和调整过程中，没有发生“阶码上溢”和“阶码下溢”的问题。因此，阶码 $E_b=0111\ 1011$。

经过上述 5 个步骤，最终得到结果为$[x+y]_浮=0\ 01111011\ 00\cdots0$。

因为$+01111011\text{B}=123$，所以阶码真值为 $123-127=-4$，尾数真值为$+1.0\cdots0\text{B}=+1.0$，所以 $x+y=+1.0\times2^{-4}=1/16=0.0625$。

从上述过程来看，本例中保留的两位附加位都起到了作用，最终都作为尾数的一部分被保留(即最终 M_b 中粗体的 **00**)，如果最初没保留这些附加位，而它们又都是非 0 值的话，则最终结果的精度就要受影响。

*3.4.2 浮点数乘除运算

在进行浮点数乘除运算前，首先应对参加运算的操作数进行判 0 处理、规格化操作和溢出判断，确定参加运算的两个操作数是正常的规格化浮点数。

浮点数乘、除运算步骤类似于浮点数加、减运算步骤，两者主要区别是，加、减运算需要对阶，而对乘、除运算来说，无需这一步。两者对结果的后处理步骤也一样，都包括规格化、舍入和阶码溢出处理。

已知两个浮点数 $X=M_x\times2^{E_x}$，$Y=M_y\times2^{E_y}$，则乘、除运算的结果如下。

$$X\times Y=(M_x\times2^{E_x})\times(M_y\times2^{E_y})=(M_x\times M_y)\times2^{E_x+E_y}$$

$$X/Y = (M_x \times 2^{E_x})/(M_y \times 2^{E_y}) = (M_x/M_y) \times 2^{E_x - E_y}$$

下面分别给出浮点数乘法和浮点数除法的运算步骤。

1. 浮点数乘法运算

假定 X 和 Y 是两个 IEEE 754 标准规格化浮点数，其相乘结果为 $M_b \times 2^{E_b}$，则求 M_b 和 E_b 的过程如下。

(1) 尾数相乘、阶码相加

尾数的乘法运算 $M_b = M_x \times M_y$ 可以采用 3.3.4 节中介绍的定点原码小数乘法算法。注意：在运算时，需要将隐藏位 1 还原到尾数中，并注意乘积的小数点位置。因为 X 和 Y 是规格化浮点数，所以其尾数 M_x 和 M_y 的真值形式都是 $\pm 1.xx\cdots x$，进行尾数乘法时，符号和数值部分分开运算，符号由 X 和 Y 两数符号异或得到，数值部分将两个形为"$1.xx\cdots x$"的定点无符号数进行 n 位数乘法运算，其结果为 $2n$ 位乘积：$xx.xx\cdots x$，小数点应该默认在第二位和第三位之间(这里的 n 取决于机器所设定的运算精度)。

阶码的相加运算 $E_b = E_x + E_y$ 采用移码相加运算算法。假设 E 为阶码，因为 IEEE 754 单精度格式浮点数的偏置常数为 127，所以，$[E]_{移} = 127 + E$。根据 IEEE 754 标准的阶码的定义，得到其阶码的加法运算规则如下。

$$\begin{aligned}[E1 + E2]_{移} &= 127 + E1 + E2 = 127 + E1 + 127 + E2 - 127 \\ &= [E1]_{移} + [E2]_{移} - 127 \\ &= [E1]_{移} + [E2]_{移} + [-127]_{补} \\ &= [E1]_{移} + [E2]_{移} + 10000001 \quad (\text{mod } 2^8)\end{aligned}$$

所以，得到阶码加法运算公式为 $[E_b]_{移} \leftarrow [E_x]_{移} + [E_y]_{移} + 129 \quad (\text{mod } 2^8)$。

例如，对于阶码分别为 10 和 -5 的两个数，用上面的计算公式计算其和的过程为 $[E_x]_{移} = 127 + 10 = 137 = 1000\ 1001B$，$[E_y]_{移} = 127 + (-5) = 122 = 0111\ 1010B$，将 $[E_x]_{移}$ 和 $[E_y]_{移}$ 代入上述公式，得 $[E_b]_{移} = [E_x]_{移} + [E_y]_{移} + 129 = 1000\ 1001 + 0111\ 1010 + 1000\ 0001\ (\text{mod } 2^8) = 1000\ 0100B = 132$，因此，其阶码的和 E_b 为 $132 - 127 = 5$，正好等于 $10 + (-5) = 5$。

(2) 尾数规格化

对于 IEEE 754 标准的规格化尾数 M_x 和 M_y 来说，一定满足以下条件：$|M_x| \geq 1$，$|M_y| \geq 1$，因此，两数乘积的绝对值应该满足 $1 \leq |M_x \times M_y| < 4$。

也就是说，数值部分得到的 $2n$ 位乘积 $xx.xx\cdots x$ 中小数点左边一定至少有一个 1，可能是 01、10、11 三种情况，若是 01，则不需要规格化；若是 10 或 11，则需要右规一次，此时，M_b 右移一位，阶码 E_b 加 1。规格化后得到的尾数数值部分的形式为 $01.xx\cdots x$，小数点左边的 1 就是隐藏位。对于 IEEE 754 浮点数的乘法运算不需要进行左规处理。

(3) 尾数舍入处理

对 $M_x \times M_y$ 规格化后得到的尾数形式为 $\pm 01.xx\cdots x$，其中小数点后面有 $(2n-2)$ 位尾数积，最终的结果肯定只能有 24 位尾数(单精度)或 53 位尾数(双精度)。因此，需要对乘积的低位部分进行舍入，其处理方法同浮点数加减运算中的舍入操作。

(4) 阶码溢出判断

在进行阶码加法、右规和舍入时，要对阶码进行溢出判断。右规和舍入时的溢出判断与

浮点数加减运算中的溢出判断方法相同。而阶码加法时的溢出判断，则要根据$[E_x]_移$、$[E_y]_移$和$[E_b]_移$最高位的取值情况进行判断，其判断规则如下：

① 若$[E_b]_移$是全1，或$[E_x]_移$和$[E_y]_移$的最高位都是1而$[E_b]_移$最高位是0，则阶码上溢。

② 若$[E_b]_移$是全0，或$[E_x]_移$和$[E_y]_移$的最高位都是0而$[E_b]_移$最高位是1，则阶码下溢。

对于情况①，若$[E_x]_移$和$[E_y]_移$的最高位都是1，说明两个阶都是正数，相加后只可能更大，并且可能大于最大阶码，若相加后$[E_b]_移$最高位是0，说明反而得到了一个负阶，那么，一定是发生了阶码上溢；对于情况②，若$[E_x]_移$和$[E_y]_移$的最高位都是0，说明两个阶都是负数，相加后只可能是更小的负阶，并且可能小于最小阶码，所以，若相加后$[E_b]_移$最高位是1，说明反而得到了一个正阶，那么，一定是发生了阶码下溢。

2. 浮点数除法运算

假定X和Y是两个IEEE 754标准规格化浮点数，其相除结果为$M_b \times 2^{E_b}$，则求M_b和E_b的过程如下。

(1) 尾数相除、阶码相减

尾数的除法运算$M_b = M_x / M_y$可以采用3.3.7节中介绍的定点原码小数除法算法。运算时需将隐藏位1还原到尾数中。因为X和Y是规格化浮点数，所以M_x和M_y的真值形式都是$\pm 1.xx\cdots x$。进行尾数相除时，符号和数值部分分开运算，符号由X和Y两数符号异或得到，数值部分将两个形为“$1.xx\cdots x$”的定点无符号数在n位无符号数除法运算部件中进行运算，其结果为n位商：$x.xx\cdots x$，小数点应该默认在第一位和第二位之间(这里的n取决于机器所设定的运算精度)。

阶码的相减运算$E_b = E_x - E_y$采用移码相减运算算法。根据IEEE 754单精度格式浮点数的阶码的定义，其阶码的减法运算规则如下。

$$\begin{aligned}[E1 - E2]_移 &= 127 + E1 - E2 = 127 + E1 - (127 + E2) + 127 \\ &= [E1]_移 - [E2]_移 + 127 \\ &= [E1]_移 + [-[E2]_移]_补 + 01111111 \quad (\bmod 2^8)。\end{aligned}$$

所以，得到阶码减法运算公式为$[E_b]_移 \leftarrow [E_x]_移 + [-[E_y]_移]_补 + 127 \quad (\bmod 2^8)$。

例如，对于阶码分别为10和−5的两个数，用上面的计算公式计算其差的过程为：

$$[E_x]_移 = 127 + 10 = 137 = 1000\ 1001B,$$

$$[E_y]_移 = 127 + (-5) = 122 = 0111\ 1010B,$$

$$[-[E_y]_移]_补 = 1000\ 0110B,$$

将$[E_x]_移$和$[E_y]_移$代入上述公式，得：

$$\begin{aligned}[E_b]_移 &= [E_x]_移 + [-[E_y]_移]_补 + 127 = 1000\ 1001 + 1000\ 0110 \\ &\quad + 0111\ 1111 (\bmod 2^8) = 1000\ 1110B = 142,\end{aligned}$$

因此，其阶码的差E_b为$142 - 127 = 15$，正好等于$10 - (-5) = 15$。

(2) 尾数规格化

对于IEEE 754标准的规格化尾数M_x和M_y来说，一定满足以下条件：

$|M_x| \geqslant 1$，$|M_y| \geqslant 1$，因此，两数相除的绝对值应该满足$1/2 \leqslant |M_x / M_y| < 2$。

也就是说，数值部分得到的n位商$x.xx\cdots x$中小数点左边的数可能是0，也可能是1。若是0，则小数点右边的第一位一定是1，此时，需要左规一次，即M_b左移一位，阶码E_b减1；若是1，则结果就是规格化形式。规格化后得到的尾数数值部分的形式为$1.xx\cdots x$，小数点

左边的 1 就是隐藏位。对于 IEEE 754 浮点数的除法运算不需要进行右规处理。

(3) 尾数舍入处理

对 M_x/M_y 规格化后得到的尾数形式为 $\pm 1.xx\cdots x$,其中小数点后面有 $n-1$ 位尾数商,因此,需要对商的低位部分进行舍入,其处理方法同浮点数加减运算中的舍入操作。

(4) 阶码溢出判断

在进行阶码减法、左规和舍入时,要对阶码进行溢出判断。左规和舍入时的溢出判断与浮点数加减运算中的溢出判断方法相同。而阶码减法时的溢出判断,则要根据 $[E_x]_移$、$[E_y]_移$ 和 $[E_b]_移$ 最高位的取值情况进行判断,其判断规则如下:

① 若 $[E_b]_移$ 是全 1,或 $[E_x]_移$ 最高位是 1 且 $[E_y]_移$ 和 $[E_b]_移$ 最高位都是 0,则阶码上溢。

② 若 $[E_b]_移$ 是全 0,或 $[E_x]_移$ 最高位是 0 且 $[E_y]_移$ 和 $[E_b]_移$ 最高位都是 1,则阶码下溢。

对于情况①,如果一个正阶减掉一个负阶得到一个负阶,那么说明一定发生阶码上溢;对于情况②,如果一个负阶减掉一个正阶得到一个正阶,那么说明一定发生阶码下溢。

3.5 运算部件的组成

前面分别介绍了各种定点数的运算方法和浮点数的运算方法,综合考虑这些运算方法后,发现所有的运算都以加减操作和移位操作为基础。因此,以一个或多个 ALU 为核心,加上移位器和存放中间临时结果的若干寄存器,在相应控制逻辑的控制下,就可以实现各种运算。所谓运算部件,通常就是指 ALU、移位器、存放临时数据的寄存器,加上用于数据选择的多路选择器和实现数据传送的总线等构成的一个运算数据通路。这种运算数据通路可以专门用一个运算器芯片来实现;也可以用若干个运算器芯片级联起来构成一个更大的运算器来实现,例如,AM2901 芯片是 4 位运算器芯片,用 4 个 AM2901 芯片可以构建一个 16 位运算器;当然,运算用的数据通路也可以和控制逻辑混合做在同一个 CPU 芯片里。

现代计算机一般都把运算部件和控制逻辑做在同一个 CPU 中,而且为了实现多条指令流水线,一个 CPU 中会有多个运算部件。广义上来说,用于执行一个特定功能的部件都可以看成是一个运算部件。如:执行定点加减运算的 ALU、执行乘(除)运算的阵列乘(除)法器、专门的存储访问部件等。通常把一个这样的运算部件称为一个执行部件或功能部件。

专门的运算器芯片或 CPU 芯片中的运算部件是计算机中数据通路的主要部分,所以,数据通路和运算器在有些场合下属于同一个概念,不同教材中可能会用不同的说法。定点运算和浮点运算不用同一套运算部件,它们是相互分开的。

*3.5.1 定点运算部件

定点运算部件用来实现无符号整数和带符号整数的各种运算,一套完整的定点运算部件,除了核心部件 ALU 外,还需要有通用寄存器组(或累加器)、多路选择器、状态/标志寄存器、移位器和用来传送数据的数据总线等。也就是说,如果要实现一个专门的定点运算器芯片,那么,该芯片中需包含 ALU、通用寄存器组、多路选择器、移位器等;如果在 CPU 中用数据通路来实现定点运算,那么,数据通路中必须包含同样的这些部件。

如图 3.31 所示是 AM2901A 的内部逻辑框图。它是一个典型的 4 位定点运算器芯片。

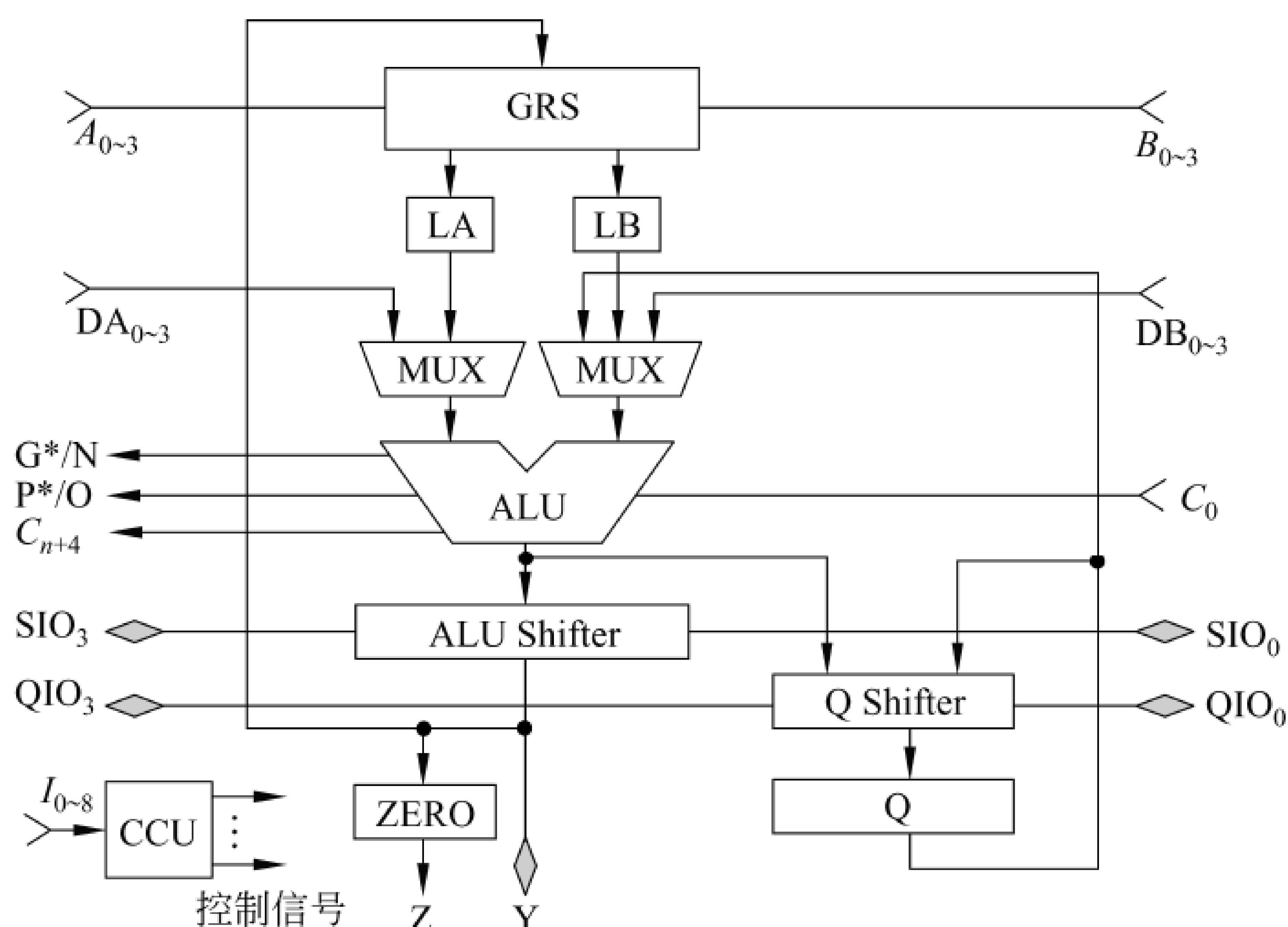


图 3.31　AM2901A 芯片的内部逻辑框图

从图 3.31 可以看出，AM2901A 的核心是一个 4 位 ALU，该 ALU 可以实现 $A+B$、$A-B$、$B-A$ 三种算术运算和“与”、“或”、“非”等 5 种逻辑运算。ALU 的功能由中央控制部件(CCU)对外部操作信号 $I_{0\sim8}$(通常是指令操作码)译码送出的控制信号进行控制。ALU 有一个进位输入信号 C_0、进位输出信号 C_{n+4}、组进位传递/溢出信号 P^*/O、组进位生成/符号信号 G^*/N。在芯片串行级联时，进位输入信号 C_0 和进位输出信号 C_{n+4} 可用来进行串行进位传递。在进行多级芯片级联时，后面两个信号 P^*/O 和 G^*/N 分别用作组进位传递信号 P^* 和组进位生成信号 G^*。对于级联中的最高一个 4 位芯片，其进位输出信号 C_{n+4} 用作进位标志，后面两个信号 P^*/O 和 G^*/N 分别用作溢出标志 O 和符号标志 N。

芯片中有一个 4 位双口通用寄存器组(General Register Set，GRS)，其中有 16 个寄存器。GRS 有一个写入口和两个读出口 A 和 B，$A_{0\sim3}$ 用来指定读出寄存器 A 的编号，$B_{0\sim3}$ 用来指定写入寄存器或读出寄存器 B 的编号。A 口和 B 口可以同时读出，分别通过锁存器 LA 和 LB 送到多路选择器 MUX 的输入端。

多路选择器 MUX 用于选择不同的操作数送 ALU 运算，ALU 的 A 输入端可能来自寄存器 A 口或从存储器来的数据 $DA_{0\sim3}$；ALU 的 B 输入端可能来自寄存器 B 口、存储器数据 $DB_{0\sim3}$ 或 Q 寄存器。MUX 的控制信号来自 CCU，通过对外部操作信号 $I_{0\sim8}$ 译码，CCU 将控制信号送到 MUX 的控制端，以确定把哪个操作数送到 ALU 运算。

该芯片中有一个 Q 寄存器，主要用于实现乘除运算。乘法运算中的部分积和除法运算中的中间余数都是双倍字长的数据，需要放到两个单倍字长寄存器中，并需要对这两个单倍字长寄存器同时串行左移(除法)或右移(乘法)。这里，Q 寄存器就是在乘法中的乘数寄存器，或是除法中的商寄存器。因此，也被称为 Q 乘商寄存器。

芯片中有两个移位寄存器，每次 ALU 运算的结果都被送到 ALU 移位寄存器中，然后和 Q 移位器一起进行左移或右移，移位后 ALU 移位器的内容送到 ALU 继续进行下次运算，而 Q 移位器的内容被送到 Q 乘商寄存器中。将 ALU 的结果进行判“0”后可通过 Z 输出端将“零”标志信息输出，Y 输入输出端可以在存储器和寄存器之间传送数据。

通过对外部操作信号 $I_{0\sim8}$ 进行译码可控制芯片完成特定的功能，其中，$I_2I_1I_0$ 三位用来控制多路选择器 MUX 以确定 ALU 操作数的输入；$I_5I_4I_3$ 三位用来控制 ALU 的操作类型；$I_8I_7I_6$ 三位用来控制 ALU 移位器和 Q 移位器的操作（左移/右移/不移）以及 Y 端数据传送方向。

AM2901A 芯片可以相互连接构成更长位数的定点运算器，例如，用 4 个 AM2901A 芯片串联可以构建一个串行进位方式的 16 位定点运算器，用一个 AM2902 芯片（CLA 部件）和 4 个 AM2901A 芯片按两级先行进位方式级联起来可以构建一个 16 位并行进位方式的定点运算器。

对于定点乘除运算来说，可以通过一个 ALU 串行执行 n 次加减运算和移位操作来实现，也可以用一个专门的阵列乘法器和阵列除法器来实现，前者速度慢，但成本低，后者速度快，但成本高。

通过类似于 AM2901 这样的通用运算器来实现乘除运算时，基本上不需要添加什么硬件逻辑，只要通过外部操作信号控制芯片正确地执行若干步骤就能完成乘除运算。假定将 R_0 用作部分积或中间余数寄存器的高位部分，Q 寄存器用作部分积或中间余数寄存器的低位部分（或商），R_1 用作被乘数或除数寄存器，则可按如下过程进行乘除运算。

(1) 预置正确的操作数。R_0 被预置为 0 或被除数的高位，Q 寄存器被预置为 0 或被除数的低位，R_1 被预置为被乘数或除数。

(2) 选择正确的 ALU 操作数。R_0 送端口 A，R_1 送端口 B。

(3) 控制 ALU 执行正确的加减操作。例如，一位乘法总是执行加操作。

(4) 选择正确的移位方式。例如，一位乘法总是右移一位。

这样，在时钟的控制下，经过若干步骤，最终就可以完成乘除运算。

目前，像 AM2901A 这样专门的运算器芯片主要用在一些特定场合，如，用来构建教学计算机或硬件实验平台等。通常情况下，一台通用计算机内部的定点运算部件都是作为 CPU 中的数据通路存在。根据执行一条指令所用时钟周期数的不同以及指令是串行执行还是流水线方式执行，可以有不同的指令执行方式，因此，相应地有不同的数据通路构建方式。主要有单周期数据通路、多周期数据通路和流水线数据通路。因为数据通路属于 CPU 的一部分，所以有关定点运算数据通路的组成与设计将在第 6 章 CPU 设计中详细介绍。

*3.5.2 浮点运算部件

20 世纪 80 年代，微处理器芯片上还集成不了很多晶体管，所以浮点数运算部件和定点数运算部件很难集成在同一块微处理器芯片中，因此，早期的机器采用了专门的浮点协处理器芯片（FPU）来执行浮点运算。如 Intel 公司早期的 8087 协处理器芯片是和 8086/8088 微处理器配套使用的，而 80287 协处理器芯片和 80286 和 80386 配套使用。早期的 MIPS 处理器也有专门的浮点协处理器。随着集成电路技术的发展，一个芯片内可实现的逻辑元件愈来愈多。因此，自从 20 世纪 90 年代以来，浮点运算部件可以直接集成在 CPU 芯片中。虽然和定点运算部件集成在一个芯片内，但两者的逻辑是分开的。也就是说 CPU 中有专门的定点运算部件和浮点运算部件。而且，存放定点数的寄存器和存放浮点数的寄存器也是分开的。几乎现在所有的处理器中都有专门的浮点数寄存器。

根据 3.4 节介绍的浮点数加减运算和浮点数乘除运算可知，阶码运算主要是移码的加

减操作；尾数运算是定点原码小数的加、减、乘、除运算。阶码运算和尾数运算分开进行，因此，阶码运算部件和尾数运算部件也是分开的。如图 3.32 所示是浮点数加减运算部件的逻辑结构示意图（虚线表示控制信号，图中省略了对两个 ALU 的控制信号线）。

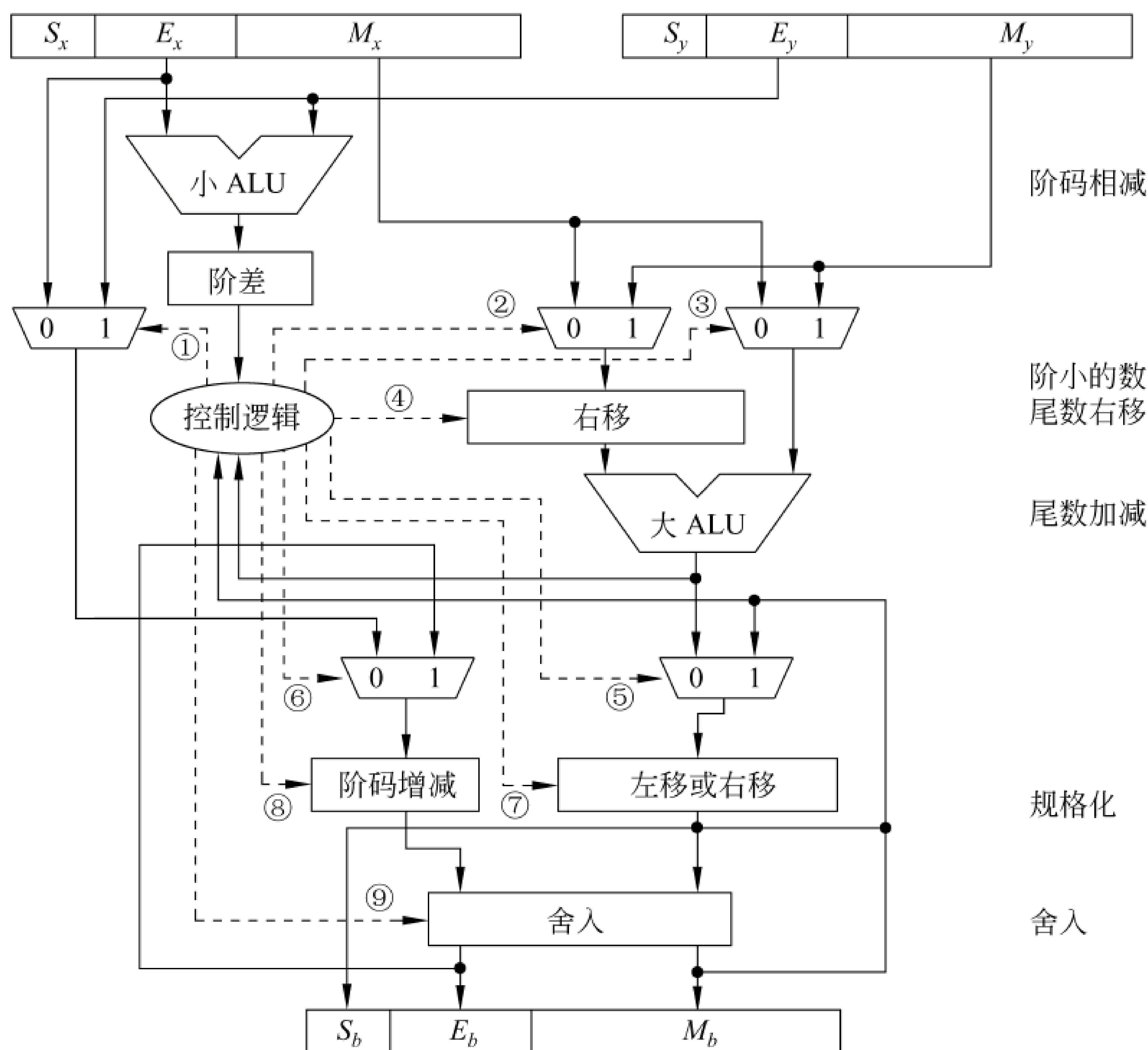


图 3.32　浮点数加减运算部件的逻辑结构

从图 3.32 看出，主要的部件有一个大 ALU 和一个小 ALU，分别执行尾数加减和阶码相减。每一步动作都由控制逻辑进行控制。

第一步：由控制逻辑控制小 ALU 实现阶码相减的操作，得到的阶差被送到控制逻辑。

第二步：由控制逻辑根据阶差的符号和绝对值来确定如何进行对阶。其中，控制信号①确定结果的阶码是 E_x 还是 E_y，②和③确定是对 M_x 还是 M_y 进行右移，④确定右移多少位。

第三步：由控制逻辑控制用对阶后的尾数在大 ALU 中进行加减，运算结果被送到控制逻辑，用于产生进行规格化的控制信号。

第四步：根据大 ALU 运算结果进行规格化。控制信号⑤和⑥确定是对大 ALU 的运算结果进行规格化还是对舍入结果进行规格化，⑦确定尾数是左移还是右移，⑧确定阶码是增加还是减少。规格化后的结果被送到舍入部件和控制逻辑。

第五步：由控制信号⑨根据规格化后的结果进行舍入，并将舍入的结果再次送到控制逻辑，以确定舍入后是否还是规格化形式，若不是，则需继续进行一次规格化。

从上述执行过程来看，浮点运算可以用流水化的形式进行。目前 CPU 中的浮点运算大多采用流水线执行方式。只要将图 3.32 所示的逻辑结构稍作调整就可以实现流水线方式的浮点运算。

对于浮点数乘除运算来说，虽然不需要对阶，但尾数的乘除运算比较复杂，并且速度较慢，所以，实现起来比加减运算复杂得多。与定点数乘除运算一样，根据机器性能/价格的不同要求，可有不同的实现方案。

3.6 十进制数加减运算

对于十进制数加减运算，通常是在二进制加减运算基础上通过适当校正来实现。计算机内实现 BCD 码加法运算时，先按二进制进行运算，然后再对结果进行修正。因为在 BCD 码加法运算中，各十进制位之间遵循十进制运算规则。十进制位与位之间是逢 10 进一，而一个十进制位的 4 位二进制码向高位进位是逢 16 进一。因此，当一个十进制数位的和大于或等于 1010(十进制的 10)时，就需要进行+6 修正。

假定 BCD 码采用 8421 码进行编码，则+6 修正的具体规则如下：

(1) 两个 BCD 码之和等于或小于 1001(即十进制的 9)时不需要修正。

(2) 两个 BCD 码之和大于或等于 1010 且小于或等于 1111(即位于十进制的 10 和 15 之间)时，需要在本位+6 修正。修正的结果是向高位产生进位。

(3) 两个 BCD 码之和大于 1111(即十进制的 15)时，会产生进位，此时需对本位进行+6 修正。

例 3.17 用十进制加法计算$(158)_{10}+(259)_{10}=(?)_{10}$。

解：

```
     0001 0101 1000
    +0010 0101 1001
  -------------------
     0011 1011 0001
    +     0110 0110
  -------------------
     0100 0001 0111        结果为 (417)10
```

十进制加法器只需要在二进制加法器上添加适当的校正逻辑就可以了。设两个 1 位十进制数 X_i 和 Y_i，它们的 8421 码分别是 $x_{i8}x_{i4}x_{i2}x_{i1}$ 和 $y_{i8}y_{i4}y_{i2}y_{i1}$，实现 X_i 和 Y_i 相加的 BCD 码加法器如图 3.33 所示。它可以看作由一个两级电路构成。第一级是一个 4 位二进制加法器，执行通常的二进制加法操作，得到 4 位二进制和 $z_{i8}^*z_{i4}^*z_{i2}^*z_{i1}^*$ 及进位输出 C_{i+1}^*。它的第二级为校正逻辑。校正逻辑方程为 $C=C_{i+1}^*+z_{i8}^*z_{i4}^*+z_{i8}^*z_{i2}^*$。

从图 3.33 中可看出，进位信号 C 可产生校正因子 0 或 6，用来对第一级电路生成的二进制和进行+6 校正。n 个一位十进制加法器可构成一个 n 位十进制串行进位加法器。

两个 BCD 码的十进制减法运算，通常采用先取减数的补码，再与被减数相加的方法实现。若相加的结果最高位产生进位，则说明结果为正数，此时，进位丢弃，余下的是正确的结果；若相加的结果最高位不产生进位，则说明结果为负数，此时，须将结果再求一次补才是正确的结果。

十进制数的补码也可采用“各位取反，末位加 1”的方法来求。通常按以下方法求 8421 BCD 码的反码。

(1) 先将 4 位 BCD 码按位取反，再加上二进制 1010(十进制 10)，加法的最高进位位丢弃。例如，BCD 码 0011(十进制 3)的反码为 1100+1010=0110(十进制 6)。

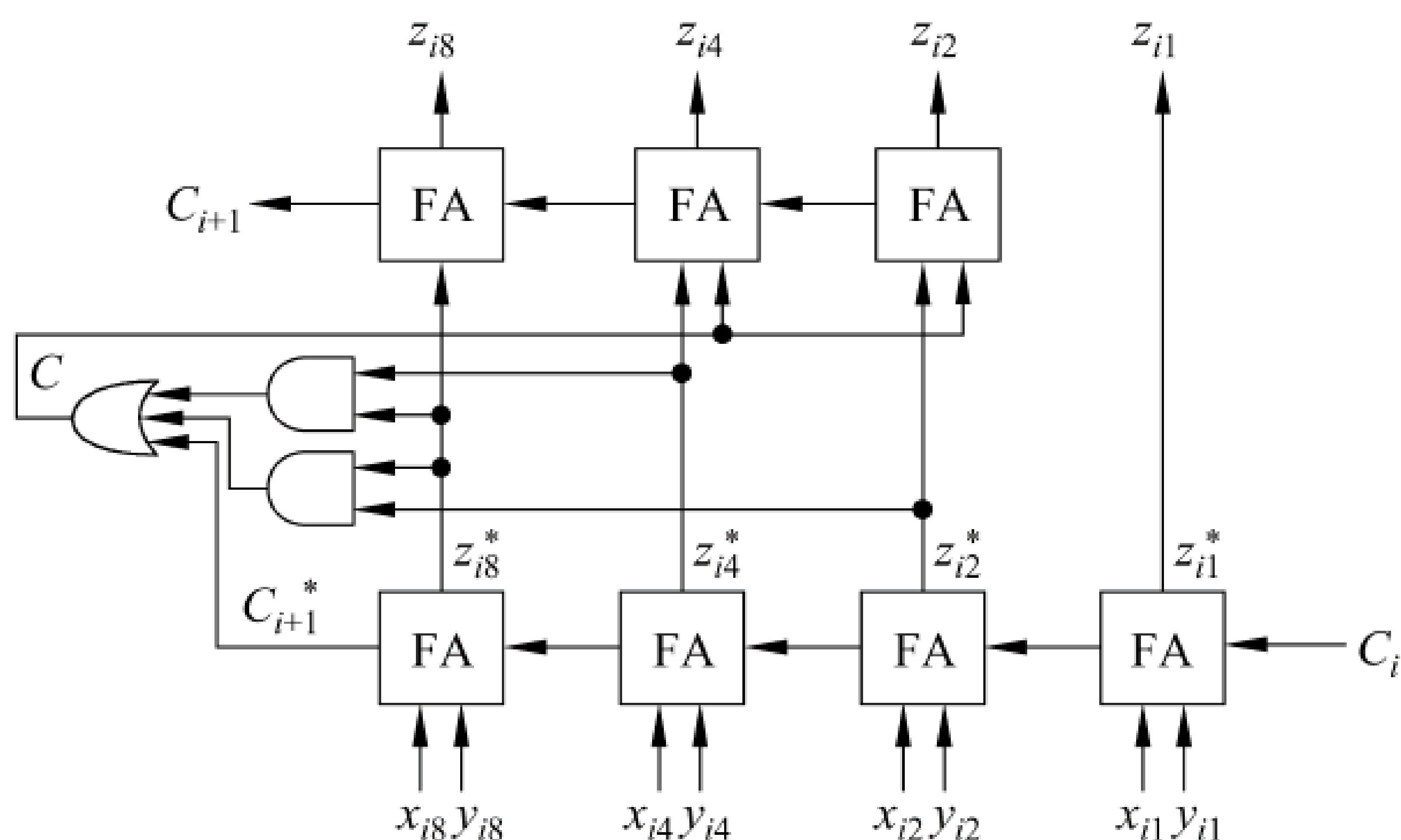


图 3.33 处理 1 位十进制数字的 BCD 码加法器

(2) 先将 4 位 BCD 码加上 0110(十进制 6),再按位取反。例如,BCD 码 0011(十进制 3)的反码是对 1001(0011+0110)按位取反的结果 0110。

例 3.18 用十进制减法计算$(158)_{10}-(259)_{10}=(?)_{10}$。

解:$[-(259)_{10}]_{补}=0111\ 0100\ 0001$

$$\begin{array}{r} 0001\ 0101\ 1000 \\ +0111\ 0100\ 0001 \\ \hline 1000\ 1001\ 1001 \end{array}$$

因为没有在最高位产生进位,所以结果为负数,求补后为$-0001\ 0000\ 0001$,故结果为$(-101)_{10}$。

3.7 本章小结

本章主要介绍计算机中涉及到的各种基本运算的算法及其实现部件。分为定点数运算和浮点数运算,它们各自用不同的运算部件实现,其中都要用到具有基本算术运算和逻辑运算功能的 ALU,而 ALU 中的核心电路是加法器,因而,快速加法器的实现是非常重要的。

本章内容具体总结如下。

- 加法器:采用并行进位方式能加快加法器速度。
  - 行波进位加法器:通过将多个一位全加器串行连接,各进位串行传递,速度慢。
  - 进位选择加法器:通过选择两个分别带进位 0 和 1 的高位部分加法器的输出来实现高、低两部分的并行执行,使运算时间减半。
  - 先行(超前)进位加法器:通过“进位生成”和“进位传递”函数来使各进位独立、并行产生,速度快。可用单级、两级或更多级先行进位方式连接。
- 算术逻辑单元 ALU:在先行进位加法器的基础上增加其他逻辑,实现基本的算术和逻辑运算的部件。有两个操作数输入、低位进位输入、一个操作控制输入、一个结果输出、一位高位进位输出和相等标志输出等。
- 定点数运算:由专门的定点运算器实现,其中核心部件是带快速加法器的 ALU。

- ◆ 移位运算
  - ▲ 逻辑移位：对无符号数进行，左(右)边补 0，低(高)位移出。
  - ▲ 算术移位：对带符号整数进行，移位前后符号位不变，否则溢出。
  - ▲ 循环移位：最左(右)边位移到最低(高)位，其他位左(右)移一位。
- ◆ 扩展运算
  - ▲ 零扩展：对无符号整数进行，高位补 0。
  - ▲ 符号扩展：对补码整数进行，在高位直接补符。
- ◆ 加减运算
  - ▲ 补码加减：用于整数加减运算。符号位和数值位一起运算，减法用加法实现。同号相加时，若结果的符号不同于加数的符号，则会发生溢出。
  - ▲ 原码加减：用于浮点数尾数加减运算。符号位和数值位分开运算，同号相加，异号相减，大数减小数，结果取大数的符号。减法用加负数补码实现。
  - ▲ 加减法器：在基本加法器基础上增加进位/加减控制、求补电路、溢出判断电路等。
- ◆ 乘法运算(用加法和右移实现)
  - ▲ 补码乘法：用于整数乘法运算，符号位和数值位一起运算，采用 Booth 算法或 MBA 算法。
  - ▲ 原码乘法：符号位和数值位分开运算，数值部分用无符号数乘法实现。
  - ▲ 快速乘法器：可用基于 CSA 的阵列乘法器、MBA＋WT 乘法器等实现。
- ◆ 除法运算(用加减法和左移实现)
  - ▲ 补码除法：符号位和数值位一起运算，有恢复余数法和不恢复余数法两种。
  - ▲ 原码除法：符号、数值分开运算，用无符号数除法实现，有恢复余数法和不恢复余数法两种。
  - ▲ 阵列除法器：用可控加减单元(CAS)组合成一个除法阵列。

● 浮点数运算(由专门的浮点运算器实现)

- ◆ 加减运算：按照以下步骤完成。
  - ▲ 对阶：向大阶看齐，阶小的尾数右移，右移时保留附加位。
  - ▲ 尾数相加减：用定点数加减运算实现，隐藏位和附加位一起参加运算。
  - ▲ 规格化处理：根据结果进行左规或右规操作。
  - ▲ 舍入：可采用就近舍入、正向舍入、负向舍入、截去 4 种方式。
  - ▲ 溢出判断：当发生阶码上溢时，结果溢出。
- ◆ 乘除运算：尾数用定点数乘除运算实现，阶码用定点数加减运算实现。

习 题 3

1. 给出以下概念的解释说明。

(1) 算术逻辑部件 ALU　　(2) 行波进位加法器　　(3) 进位选择加法器
(4) 先行(超前)进位加法器　　(5) 成组先行进位　　(6) 零标志 ZF

(7) 溢出标志 OF　　(8) 进位/借位标志 CF　　(9) 符号标志 NF
(10) 布斯乘法　　(11) 改进布斯算法 MBA(基 4 Booth 算法)
(12) 阵列乘法器　　(13) 进位保存加法器 CSA　　(14) 对阶
(15) 舍入　　(16) 保护位　　(17) 舍入位
(18) 粘位　　(19) 规格化浮点数　　(20) 右规
(21) 左规　　(22) 阶码上溢　　(23) 阶码下溢
(24) 数据通路　　(25) 多路选择器　　(26) 桶型移位器
(27) 执行部件　　(28) 功能部件　　(29) 通用寄存器组 GRS
(30) Q 乘商寄存器　　(31) 状态/标志寄存器

2. 简单回答下列问题。

(1) 为何在高级语言和机器语言中都要提供“按位运算”? 为何高级语言需要提供逻辑运算? 按位运算和逻辑运算的差别是什么?

(2) 如何进行逻辑移位和算术移位? 它们各用于哪种类型的数据?

(3) 移位运算和乘除运算具有什么关系?

(4) 高级语言中的运算和机器语言(即指令)中的运算是什么关系? 假定某一个高级语言源程序 P 中有乘、除运算,但机器 M 中不提供乘、除运算指令,则程序 P 能否在机器 M 上运行? 为什么?

(5) 为什么用 ALU 和移位器就能实现定点数和浮点数的所有加、减、乘、除运算?

(6) 影响加法运算速度的关键问题有哪些? 可采取什么措施? 对于乘法运算呢?

(7) 能否用快速乘法器实现除法运算? 如何实现?

3. 考虑以下 C 语言程序代码:

```
int func1(unsigned word)
{
    return (int) ((word<<24) >>24);
}
int func2(unsigned word)
{
    return ((int) word <<24) >>24;
}
```

假设在一个 32 位机器上执行这些函数,该机器使用二进制补码表示带符号整数。无符号数采用逻辑移位,带符号整数采用算术移位。请填写表 3.6,并说明函数 func1 和 func2 的功能。

表 3.6　题 3 用表

w		func1(w)		func2(w)	
机器数	值	机器数	值	机器数	值
	127				
	128				
	255				
	256				

4. 填写表 3.7,注意对比无符号数和带符号整数的乘法结果以及截断操作前后的结果。

表 3.7 题 4 用表

模式	x		y		$x \times y$（截断前）		$x \times y$（截断后）	
	机器数	值	机器数	值	机器数	值	机器数	值
无符号数	110		010					
二进制补码	110		010					
无符号数	001		111					
二进制补码	001		111					
无符号数	111		111					
二进制补码	111		111					

5. 以下是两段 C 语言代码，函数 arith()是直接用 C 语言写的，而 optarith()是对 arith()函数以某个确定的 M 和 N 编译生成的机器代码反编译生成的。根据 optarith()，可以推断函数 arith()中 M 和 N 的值各是多少？

```
#define M
#define N
int arith (int x, int y)
{
    int result=0 ;
    result=x * M+ y/N;
    return result;
}

int optarith (int x, int y)
{
    int t=x;
    x<<=4;
    x-=t;
    if (y<0) y+=3;
    y>>=2;
    return x+y;
}
```

6. 设 $A_4 \sim A_1$ 和 $B_4 \sim B_1$ 分别是 4 位加法器的两组输入，C_0 为低位来的进位。当加法器分别采用串行进位和先行进位时，写出 4 个进位 $C_4 \sim C_1$ 的逻辑表达式。

7. 用 SN74181 和 SN74182 器件设计一个 16 位先行进位补码加减运算器，画出运算器的逻辑框图，并给出零标志、进位标志、溢出标志、符号标志的生成电路。

8. 用 SN74181 和 SN74182 器件设计一个 32 位的 ALU，要求采用两级先行进位结构。

(1) 写出所需的 SN74181 和 SN74182 芯片数。

(2) 画出 32 位 ALU 的逻辑结构图。

9. 已知 $x=10, y=-6$，采用 6 位机器数表示。请按如下要求计算，并把结果还原成真值。

(1) 求$[x+y]_{补}$，$[x-y]_{补}$。

(2) 用原码一位乘法计算$[x \times y]_{原}$。

(3) 用 MBA(基 4 布斯)乘法计算$[x \times y]_{补}$。

(4) 用不恢复余数法计算$[x/y]_{原}$的商和余数。

(5) 用不恢复余数法计算$[x/y]_{补}$的商和余数。

10. 若一次加法需要1ns,一次移位需要0.5ns。请分别计算用一位乘法、两位乘法、基于CRA的阵列乘法、基于CSA的阵列乘法4种方式计算两个8位无符号二进制数乘积时所需的时间。

11. 在IEEE 754浮点数运算中,当结果的尾数出现什么形式时需要进行左规,什么形式时需要进行右规?如何进行左规,如何进行右规?

12. 在IEEE 754浮点数运算中,如何判断浮点运算的结果是否溢出?

13. 假设浮点数格式为:阶码是4位移码,偏置常数为8,尾数是6位补码(采用双符号位),用浮点运算规则分别计算在不采用任何附加位和采用两位附加位(保护位、舍入位)两种情况下的值(假定对阶和右规时采用就近舍入到偶数方式)。

(1) $(15/16)\times2^7+(2/16)\times2^5$　　(2) $(15/16)\times2^7-(2/16)\times2^5$

(3) $(15/16)\times2^5+(2/16)\times2^7$　　(4) $(15/16)\times2^5-(2/16)\times2^7$

14. 采用IEEE 754单精度浮点数格式计算下列表达式的值。

(1) 0.75+(-65.25)　　(2) 0.75-(-65.25)

15. 假定十进制数用8421 NBCD码表示,采用十进制加法运算计算下列表达式的值,并讨论在十进制BCD码加法运算中如何判断溢出。

(1) 234+567　　(2) 548+729

16. 假定十进制数用8421 NBCD码表示,十进制运算673-356可以采用673加上-356的模10补码实现。画出实现上述操作的3位十进制数的BCD码减法运算线路,列出线路中所有的输入变量和输出变量。

第 4 章 存储器分层体系结构

存储器是计算机系统的重要组成部分,用来存放程序和数据。有了存储器,计算机就有了记忆能力,从而能自动地从存储器中取出保存的指令按序进行操作。计算机中所用的记忆元件有多种类型,如寄存器、静态 RAM、动态 RAM、磁盘、磁带、光盘等,它们各自有不同的速度、容量和价格,各类存储器按照层次化方式构成存储器分层体系结构。

本章主要介绍构成存储器分层体系结构的几类存储器的工作原理和组织形式。包括半导体随机存取存储器、只读存储器和 Flash 存储器的基本读写原理和组织结构,存储器芯片和 CPU 的连接,并行存储器结构,高速缓存的基本原理和实现技术,以及虚拟存储器系统的实现技术。

4.1 存储器概述

4.1.1 存储器的分类

根据存储器的特点和使用方法的不同,可以有以下几种分类方法。

1. 按存储元件分类

存储元件必须具有两个截然不同的物理状态,才能被用来表示二进制代码 0 和 1。目前使用的存储元件主要有半导体器件、磁性材料和光介质。用半导体器件构成的存储器称为半导体存储器;磁性材料存储器主要是磁表面存储器,如磁盘存储器和磁带存储器;光介质存储器称为光盘存储器。

2. 按存取方式分类

(1) 随机存取存储器(RAM)

随机存取存储器(Random Access Memory)的特点是按地址访问存储单元,因为每个地址译码时间相同,所以,在不考虑芯片内部缓冲的前提下,每个单元的访问时间是一个常数,与地址无关。不过,现在的 DRAM 芯片内都具有行缓冲,因而有些数据可能因为已在缓冲而缩短了访问时间。半导体存储器属于随机存取存储器,可用做 cache 和主存储器。

(2) 顺序存取存储器(SAM)

顺序存取存储器(Sequential Access Memory)的特点是信息按顺序存放和读出,其存取时间取决于信息存放位置,以记录块为单位编址。磁带存储器就是一种顺序存取存储器,其存储容量大,但存取速度慢。

(3) 直接存取存储器(DAM)

直接存取存储器(Direct Access Memory)的存取方式兼有随机访问和顺序访问的特点。首先可直接选取所需信息所在区域,然后按顺序方式存取,磁盘存储器就是如此。

(4) 相联存储器(CAM)

上述三类存储器都是按所需信息的地址来访问,但有些情况下可能不知道所访问信息的地址,只知道要访问信息的内容特征,此时,只能按内容检索到存储位置进行读写。这种存储器称为按内容访问存储器(Content Addressed Memory)或相联存储器(Associative Memory)。如快表就是一种相联存储器。

3. 按信息的可更改性分类

按信息的可更改性分为读写存储器(Read/Write Memory)和只读存储器(Read Only Memory)。读写存储器中的信息可以读出和写入,RAM 芯片是一种读写存储器;只读存储器用 ROM 表示,ROM 存储器芯片中的信息一旦确定,通常情况下只读不写,但在某些情况下也可重新写入。

RAM 芯片和 ROM 芯片都采用随机存取方式进行信息的访问。

4. 按断电后信息的可保存性分类

按断电后信息的可保存性分成非易失(不挥发)性存储器(Nonvolatile Memory)和易失(挥发)性存储器(Volatile Memory)。非易失性存储器的信息可一直保留,不需电源维持。如 ROM、磁表面存储器、光存储器等;易失性存储器在电源关闭时信息自动丢失。如 RAM、cache 等。

5. 按功能分类

(1) 高速缓冲存储器

目前高速缓存(cache)由静态 RAM 芯片组成,位于主存和 CPU 之间,存取速度接近 CPU 的工作速度,用来存放当前 CPU 经常使用到的指令和数据。

(2) 主存储器

指令直接面向的存储器是主存储器,简称主存。CPU 执行指令时给出的存储器地址是主存地址(虚拟存储系统中,需要将指令给出的逻辑地址转换成主存地址)。因此,主存是存储器分层体系结构中的核心存储器,用来存放系统中被启动运行的程序及其数据,主存目前一般用 MOS 管半导体存储器构成。

(3) 辅助存储器

把系统运行时直接和主存交换信息的存储器称为辅助存储器,简称辅存。磁盘存储器相对于磁带和光盘存储器速度快,因此,目前大多用磁盘存储器作为辅存,辅存的内容需要调入主存后才能被 CPU 访问。

(4) 海量后备存储器

磁带存储器和光盘存储器的容量大、速度慢,主要用于信息的备份和脱机存档,因此被用作海量后备存储器。

辅存和海量后备存储器统称为外部存储器,简称外存。

4.1.2 主存储器的组成和基本操作

如图 4.1 所示是主存储器(Main Memory,MM)的基本框图。其中由一个个存储 0 或

1 的记忆单元(cell)构成的存储阵列是存储器的核心部分。这种记忆单元也称为存储元、位元,它是具有两种稳态的能表示二进制 0 和 1 的物理器件。存储阵列(bank)也被称为存储体、存储矩阵。为了存取存储体中的信息,必须对存储单元编号,所编号码就是地址。编址单位(addressing unit)是指具有相同地址的那些位元构成的一个单位,可以是一个字节或一个字。对各存储单元进行编号的方式称为编址方式(addressing mode),可以按字节编址,也可以按字编址。现在大多数通用计算机都采用字节编址方式,此时,存储体内一个地址中有一个字节。也有许多专用于科学计算的大型计算机采用 64 位编址,这是因为科学计算中数据大多是 64 位浮点数。

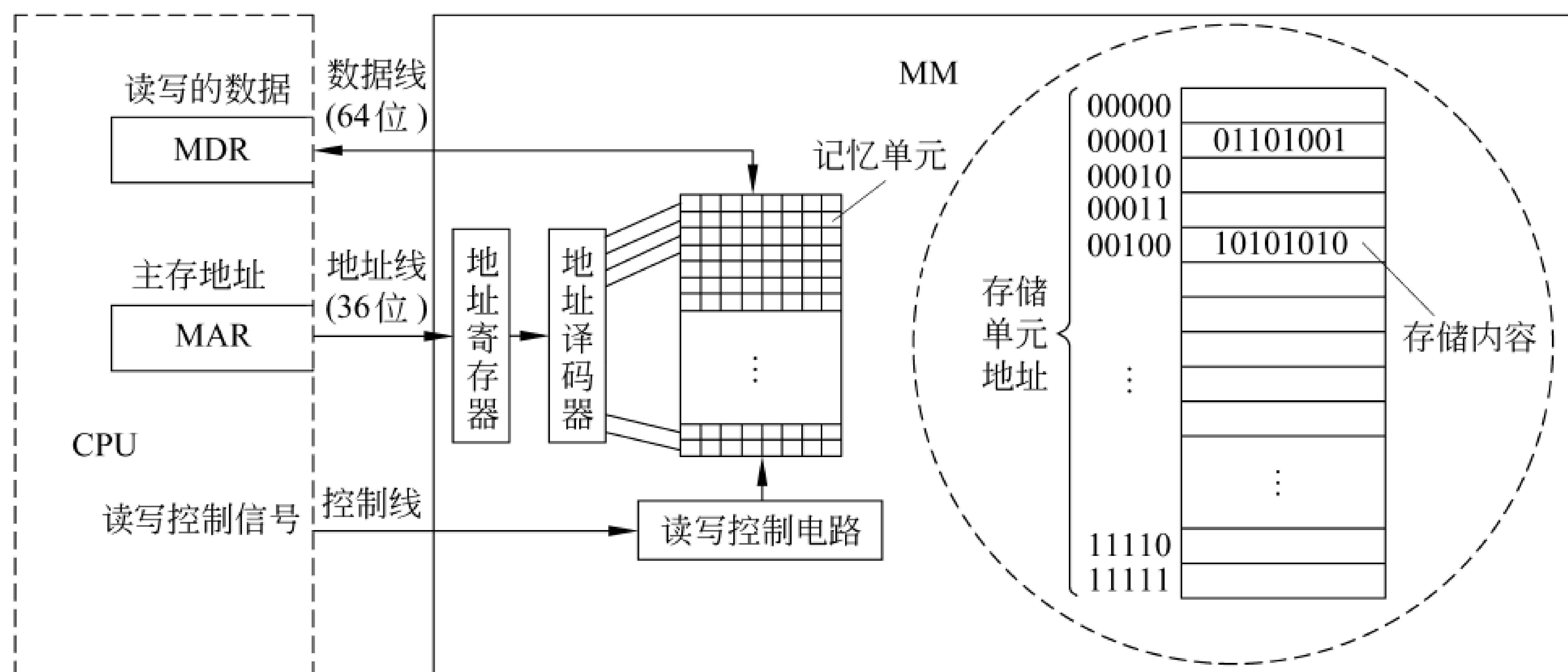


图 4.1　主存储器基本框图

指令执行过程中需要访问主存时,CPU 首先把被访问单元的地址送到主存地址寄存器 MAR(Memory Address Register)中,然后通过地址线将主存地址送到主存中的地址寄存器,以便地址译码器进行译码选中相应单元,同时,CPU 将读写信号通过控制线送到主存的读写控制电路。如果是写操作,CPU 同时将要写的信息送主存数据寄存器 MDR(Memory Data Register)中,在读写控制电路的控制下,经数据线将信息写入选中的单元;如果是读操作,则主存读出选中单元的内容送数据线,然后被送到 MDR 中。数据线的宽度与 MDR 的宽度相同,地址线的宽度与 MAR 的宽度相同。图 4.1 中采用 64 位数据线,所以在字节编址方式下,每次最多可以存取 8 个字节的内容。地址线的位数决定了主存地址空间的最大可寻址范围,例如,36 位地址的最大寻址范围为 $0\sim2^{36}-1$。

4.1.3　存储器的主要性能指标

虽然在计算机出现至今的几十年内,存储器介质和特性有了很大变化,但评价其性能的主要指标仍然是容量、速度和价格。

存储器容量指它能存放的二进制位数或字(字节)数;存储器的价格可用总价格 C 或每位价格 c 来表示,若存储器按位计算的容量为 S,则 $c=C/S$。总价 C 中应包括存储单元本身的价格以及完成存储器操作所必需的外围电路的价格;存储器速度可用访问时间、存储周期或带宽来表示。

访问时间一般用读出时间 T_A 及写入时间 T_W 来描述。T_A 是指从存储器接到读命令

开始至信息被送到数据线上所需的时间；T_W 是指存储器接到写命令开始至信息被写入存储器所需的时间。

存储周期是指存储器进行一次读写操作所需要的全部时间，也就是存储器进行连续读写操作所允许的最短间隔时间。它应等于访问时间加上下一次存取开始前所要求的附加时间，一般用 T_M 表示。存储器中由于读出放大器、驱动电路等都有一段稳定恢复时间，读出后不能立即进行下一次访问。所以，一般 T_M、T_A 和 T_W 存在以下关系：$T_M>T_A$、$T_M>T_W$。

存储器的带宽 B 表示存储器被连续访问时，可以提供的数据传送速率，通常用每秒钟传送信息的位数(或字节数)来衡量。

随着集成电路制造工艺和集成度的不断提高，在过去的 10 年中存储器的容量大致以每两年翻一番的速度增长；而存储器价格则以每年 35% 的速度下降，从 1990 年的约 100 美元/MB，下降到 2000 年约 1 美元/MB，到 2009 年更降至 0.01 美元/MB；在访问性能方面，存储器的访问延迟性能以每年约 7% 的速度提高，目前 DDR SDRAM 的访问延时约在 10～20ns 之间；在传输数据带宽方面，目前市场流行的 DDR2 SDRAM 其单芯片数据带宽可达 600～800MBps，而其存储模块的数据带宽可达 5～6GBps。

4.1.4 存储器的层次化结构

存储器容量和性能应随着处理器速度和性能的提高而同步提高，以保持系统性能的平衡。然而，在过去 20 多年中，随着时间的推移，处理器和存储器在性能发展上的差异越来越大，存储器在容量尤其是访问延时方面的性能增长越来越跟不上处理器性能发展的需要。为了缩小存储器和处理器两者之间在性能方面的差距，通常在计算机内部采用层次化的存储器体系结构。

因为某一种元件制造的存储器很难同时满足大容量、高速度和低成本的要求。比如双极型半导体存储器的存取速度快，但是难以构成大容量存储器。而大容量、低成本的磁表面存储器的存取速度又远低于半导体存储器，并且难以实现随机存取。因此，在计算机中把各种不同容量和不同存取速度的存储器按一定的结构有机地组织在一起，形成层次化的存储器体系结构。程序和数据按不同的层次存放在各级存储器中，整个存储系统在速度、容量和价格等方面具有较好的综合性能指标，图 4.2 是存储系统层次结构示意图。

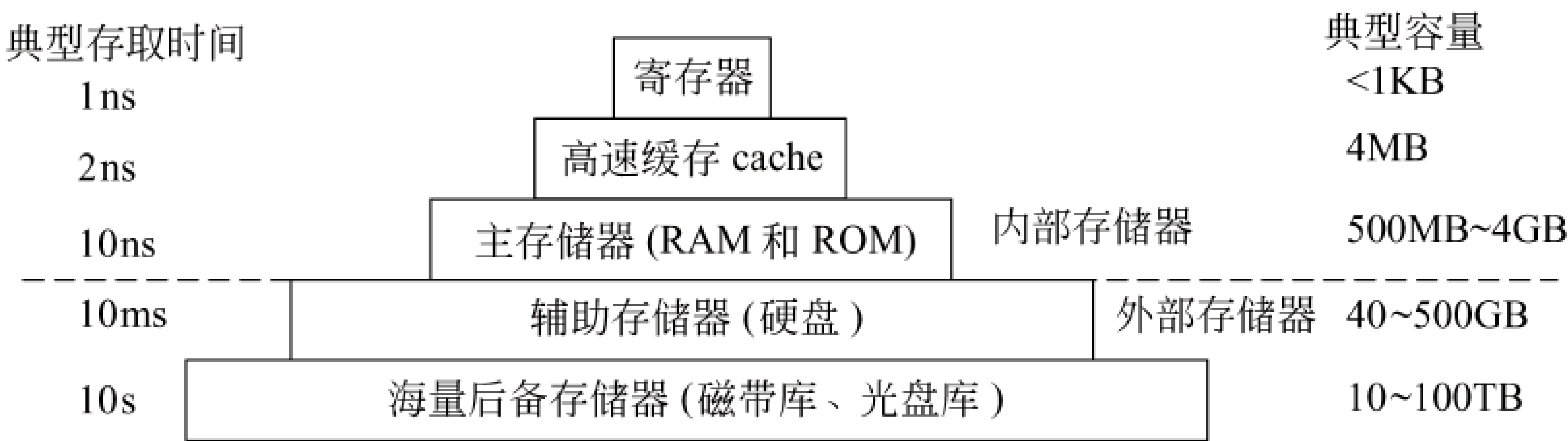


图 4.2 存储器层次化体系结构示意图

虽然图 4.2 中给出的典型存取时间和存储容量会随时间变化，但这些数据反映了速度和容量之间的关系，以及层次化结构存储器的构成思想。速度越快则容量越小、越靠近 CPU。CPU 可以直接访问内部存储器，而外部存储器的信息则要先取到主存，然后才能被 CPU 访问。

数据一般只在相邻两层之间复制传送，而且总是从慢速存储器复制到快速存储器被使用。传送的单位是一个定长块，因此需要确定定长块的大小，并在相邻两层间建立块映射关系。

CPU 执行指令时，需要的操作数大部分都来自寄存器。如果需要从(向)存储器中取(存)数据时，先访问 cache，如果不在 cache，则访问主存，如果不在主存，则访问硬盘，此时，操作数从硬盘中读出送到主存，然后从主存送到 cache。

4.2 半导体随机存取存储器

半导体读写存储器简称 RWM，习惯上也称为 RAM。半导体 RAM 具有体积小、存取速度快等优点，因而适合作为内部存储器使用。按工艺不同可将半导体 RAM 分为双极型 RAM 和 MOS 型 RAM 两大类，MOS 型 RAM 又分为静态 RAM(Static RAM，SRAM)和动态 RAM(Dynamic RAM，DRAM)。

4.2.1 基本存储元件

基本存储元件用来存储一位二进制信息，是组成存储器的最基本的电路。下面介绍两种典型的分别用于 SRAM 芯片和 DRAM 芯片的存储元件。

1. 六管静态 MOS 管存储元件

如图 4.3 所示，其中 T_1 和 T_2 构成触发器，T_5、T_6 是触发器的负载管，T_3、T_4 为门控管。使用这 6 个 MOS 管即可组成存储一位二进制信息的基本存储元。若 T_2 管导通，则 T_1 管一定截止，此时 A 点为高电平，B 电为低电平，假定此时为存“1”状态；反之(当 T_1 管导通时)则为存“0”状态。

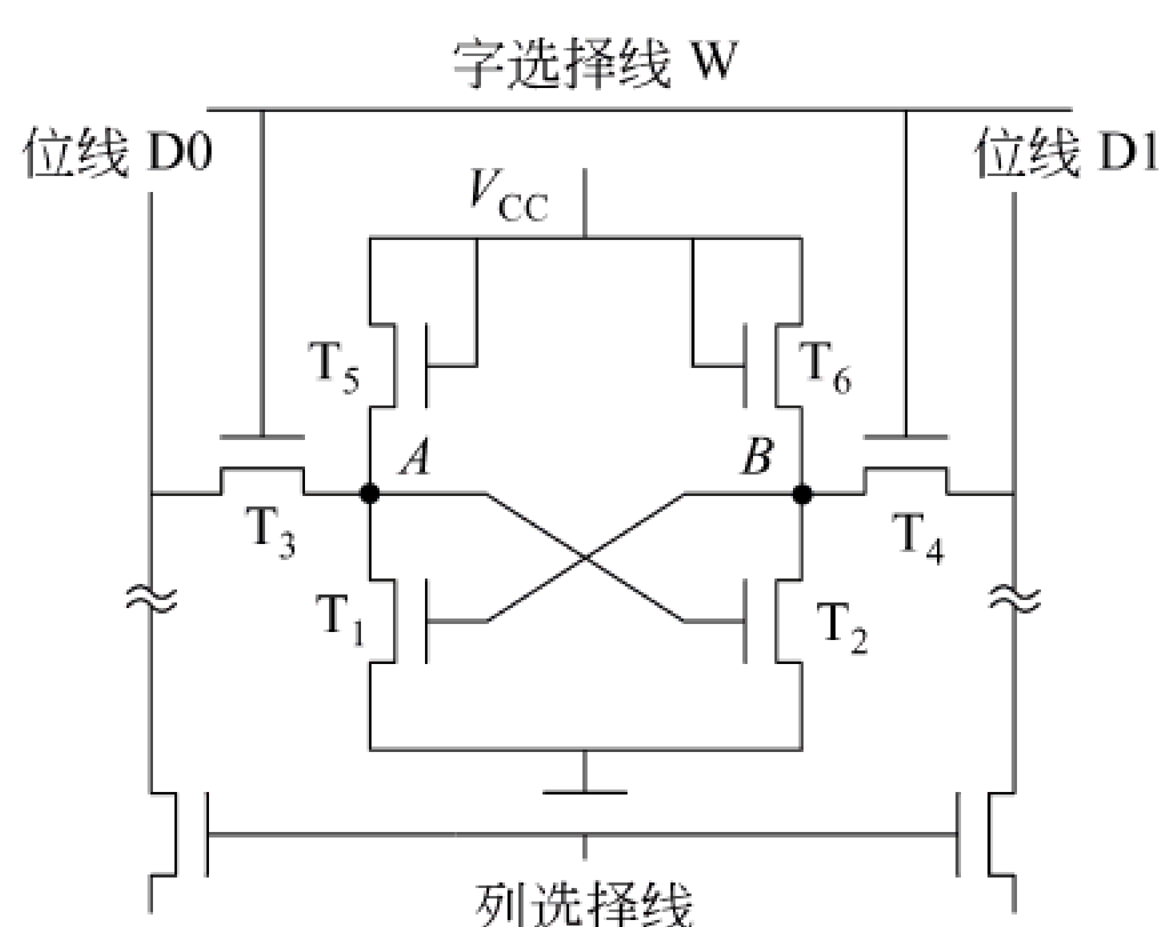


图 4.3　6 管静态存储元件

(1) 信息的保持

字选择线 W 加低电平，T_3 与 T_4 截止，触发器与外界隔离，保持原有信息不变。

(2) 读出

首先在两条位线上加高电平，当字选择线上加高电平时，T_3 与 T_4 开启。若原存“0”，则 B 点为高电平，T_1 管导通，因而有电流从位线 D0 经 T_3、T_1 管流到地，从而在位线 D0 上产生一个负脉冲，位线 D1 上没有负脉冲。反之，若原存“1”，则在位线 D1 上有负脉冲。根据哪条位线上有负脉冲可区分读出的是“0”还是“1”。

(3) 写入

字选择线上加高电平，T_3 与 T_4 开启。若要写“1”，则在位线 D1 上加低电平，使 B 点电位下降，T_1 管截止，A 点电位上升，使 T_2 管导通完成写“1”。若要写“0”，则在 D0 线上加低电平，使 A 点电位下降，结果使 T_2 管截止，B 点电位上升，T_1 管导通，完成写“0”。

2. 单管动态 MOS 管存储元件

从 6 管静态 RAM 电路可看出，即使存储元件不工作，也有电流流过。如 T_1 导通，T_2

截止时，有从电源经 $T_5 \rightarrow T_1 \rightarrow$ 地的电流流动。反之，则有从电源经 $T_6 \rightarrow T_2 \rightarrow$ 地的电流流动，因而功耗较大。

动态 RAM 利用 MOS 管的栅极电容来保存信息，在信息保持状态下，存储元件中没有电流流动，因而大大降低了功耗。

DRAM 芯片中一般采用图 4.4 所示的单管动态单元电路，其中 T 管为字选门控管，读写时加选通脉冲使其导通。

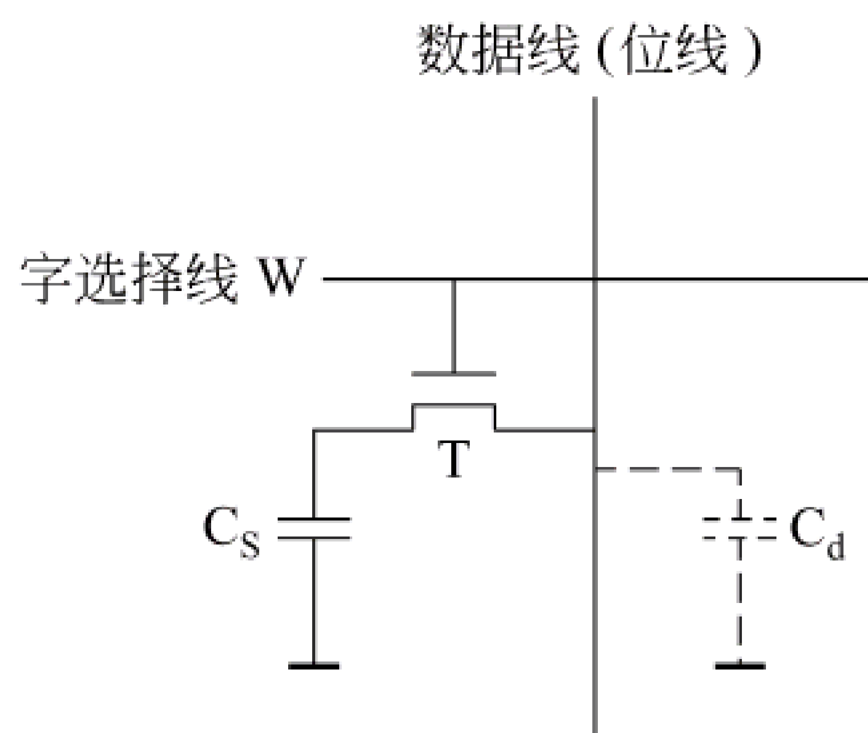


图 4.4　单管动态存储元件

(1) 读出。若原存“1”，则 C_S 上电荷通过 T 管在数据线上产生电流。反之，若原存“0”，则无电流。由此可区分读出的是 0 还是 1。因为读出时 C_S 上电荷放电，电位下降，所以是破坏性读出，读后应有重写操作，称为“再生”。由于 C_S 不可能很大，所以 C_S 在数据线上放电产生的电流不会很大，而且由于寄生电容 C_d 的存在，放电时 C_S 上的电荷是在 C_S 和 C_d 之间分配，因此，读出电流值实际上非常小，故对读出放大器的要求较高。

(2) 写入。写“1”时，在数据线上加高电平，经 T 管对 C_S 充电；写“0”则在数据线上加低电平，C_S 充分放电而使其上无电荷。

(3) 刷新(refresh)。由于 MOS 管栅极上存储的电荷会缓慢放电，超过一定时间，就会丢失信息。因此必须定时给栅极电容充电，这一过程称为“刷新”。

3. 静态存储元件和动态存储元件的比较

根据以上对两种典型 SRAM 元件和 DRAM 元件的介绍可看出以下情况。

SRAM 存储元件所用 MOS 管多，占硅片面积大，因而功耗大，集成度低；但因为采用一个正负反馈触发器电路来存储信息，所以，只要直流供电电源一直加在电路上，就能一直保持记忆状态不变，所以，无须刷新；也不会因为读操作而使状态发生改变，故无须读后再生；特别是它的读写速度快，其存储原理可看作是对带时钟的 RS 触发器的读写过程。由于 SRAM 价格比较昂贵，因而，适合做高速小容量的半导体存储器，如 cache。

DRAM 存储元件所用 MOS 管少，占硅片面积小，因而功耗小，集成度很高；但因为采用电容储存电荷来存储信息，会发生漏电现象，所以要使状态保持不变，必须定时刷新；因为读操作会使状态发生改变，故需读后再生；特别是它的读写速度相对 SRAM 元件要慢得多，其存储原理可看作是对电容充、放电的过程。相比于 SRAM，DRAM 价格较低，因而适合做慢速大容量的半导体存储器，如主存。

4.2.2　静态 RAM 芯片

1. SRAM 芯片结构

如图 4.5 所示，静态 MOS 存储器芯片由存储体、I/O 读写电路、地址译码和控制电路等部分组成。

(1) 存储体(存储矩阵)。是存储单元的集合。如图 4.5 所示，4096 个存储单元被排成 64×64 的存储阵列，称为位平面，这样 8 个位平面构成 4096 字节的存储体。由 X 选择线(行选择线)和 Y 选择线(列选择线)来选择所需单元，不同位平面的相同行、列上的位同时被读出或写入。



图 4.5　SRAM 存储器芯片结构图

(2) 地址译码器。用来将地址转换为译码输出线上的高电平，以便驱动相应的读写电路。地址译码有一维译码和二维译码两种方式。一维方式也称为线选法或单译码法，适用于小容量存储器，二维方式也称为重合法或双译码法，适用于容量较大的存储器。

在单译码方式下，只有一个行译码器，同一行中所有存储单元的字线连在一起，接到地址译码器的输出端，这样，选中一行中的各单元构成一个字，被同时读出或写入，这种结构的存储器芯片被称为字片式芯片。

地址位数较多时，地址译码器输出线太多。比如，$n=12$ 时单译码结构要求译码器有 4096 根输出线(字选择线)，因此，大容量存储器芯片不宜采用一维单译码方式的字片式芯片结构。

目前，存储芯片大多采用双译码结构。地址译码器分为 X 和 Y 方向两个译码器。图 4.5 采用的就是二维双译码结构，其存储阵列组织如图 4.6 所示。



图 4.6　二维双译码结构(位片式芯片)

图 4.6 中的存储阵列有 4096 个单元，需要 12 根地址线：$A_0 \sim A_{11}$，其中 $A_0 \sim A_5$ 送至 X 译码器，有 64 条译码输出线，各选择一行单元；$A_6 \sim A_{11}$ 送至 Y 译码器，它也有 64 条译码输出线，分别控制一列单元的位线控制门。假如输入的 12 位地址为 $A_{11}A_{10}\cdots A_0 =$ 0000 0000 0001，则 X 译码器的第 2 根译码输出线（X_1）为高电平，于是与它相连的 64 个存储单元的字选择 W 线为高电平。Y 译码器的第 1 根译码输出线（Y_0）为高电平，打开第一列的位线控制门。在 X、Y 译码的联合作用下，存储矩阵中（1，0）单元被选中。

在选中的行和列交叉点上的单元只有一位，因此，采用二维双译码结构的存储器芯片被称为位片式芯片。有些芯片的存储阵列采用三维结构，用多个位平面构成存储阵列，不同位平面在同一行和列交叉点上的多位构成一个存储字，被同时读出或写入。

（3）驱动器。在双译码结构中，一条 X 方向的选择线要控制在其上的各个存储单元的字选择线，所以负载较大，因此需要在译码器输出后加驱动器。

（4）I/O 控制电路。用以控制被选中的单元的读出或写入，具有放大信息的作用。

（5）片选控制信号。单个芯片容量太小，往往满足不了计算机对存储器容量的要求，因此需将一定数量的芯片按特定方式连接成一个完整的存储器。在访问某个字时，必须“选中”该字所在芯片，而其他芯片不被“选中”。因而芯片上除了地址线和数据线外，还应有片选控制信号。在地址选择时，由芯片外的地址译码器的输入信号以及控制信号，如“访存控制”来产生片选控制信号，选中要访问的存储字所在的芯片。

（6）读写控制信号。根据 CPU 给出的是读命令还是写命令，控制被选中存储单元的读或写。

2. SRAM 芯片读写周期

要保证正确读写，必须注意 CPU 时序与存储器读写周期的配合。一般存储器芯片手册都给出了芯片读写周期的时序图，图 4.7 是 SRAM 读写周期时序示意图。

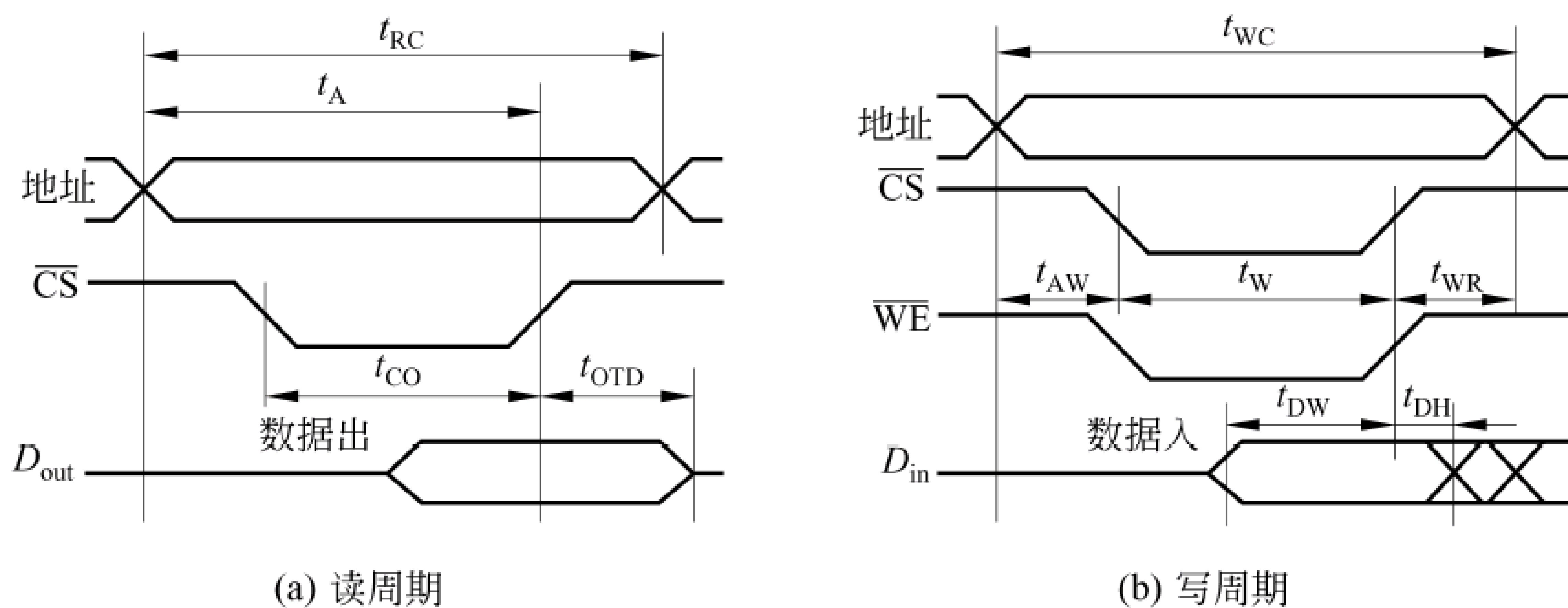


图 4.7　SRAM 读写周期时序图

在读周期中，地址线先有效，以便进行地址译码。为了读出数据，必须保证片选信号 $\overline{CS}$ 有效（低电平），$\overline{WE}$ 信号为高。从地址有效开始，经过 t_A（读出时间），数据在外部数据线上稳定出现。t_{CO} 表示从 $\overline{CS}$ 有效到数据稳定出现在外部数据线上的时间。t_{OTD} 为片选信号无效后数据还能保持的时间。t_{RC}（读周期）代表连续两次读操作之间的最小间隔时间，$t_A \leqslant t_{RC}$。

在写周期中，$\overline{CS}$和$\overline{WE}$信号都应该有效，即都是低电平。t_W 为写入时间，为保证数据可靠地写入，$\overline{CS}$与$\overline{WE}$同时有效的时间必须大于或等于 t_W。地址有效后，必须经过 t_{AW} 时间，$\overline{WE}$信号才能有效，否则可能产生写出错。$\overline{WE}$无效后，经 t_{WR} 时间地址才能改变，否则也可能错误地写入。写入数据必须在$\overline{WE}$和$\overline{CS}$无效之前 t_{DW} 时间就送到数据线上。t_{DH} 为$\overline{WE}$无效后数据还要保持的时间。t_{WC} 为写周期，表示连续两次写操作之间的最小时间间隔，$t_W \leqslant t_{WC}$。

4.2.3 动态 RAM 芯片

1. DRAM 芯片结构

图 4.8 是典型的 4M×4 位 DRAM 芯片示意图。DRAM 芯片容量较大，因而地址位数较多，为了减少芯片的地址引脚数，从而减小体积，大多采用地址引脚复用技术，行地址和列地址通过相同的管脚分先后两次输入，这样地址引脚数可减少一半。

图 4.8(a)给出了芯片的引脚，共有 11 根地址引脚线 $A_0 \sim A_{10}$，在行选通信号$\overline{RAS}$和列选通信号$\overline{CAS}$的控制下分时传送行、列地址，有 4 根数据引脚线 $D_1 \sim D_4$，因此，每个芯片同时读出 4 位数据，$\overline{WE}$为读写控制引脚，低电平时为写操作；$\overline{OE}$为输出使能驱动引脚，低电平有效，高电平时断开输出。

图 4.8(b)给出了芯片内部的逻辑结构图，芯片存储阵列采用三维结构，芯片容量为 2048×2048×4 位，因此，行地址和列地址各 11 位，有 4 个位平面，在每个行、列交叉处的 4 个位平面数据同时进行读写。行地址缓冲器和刷新计数器通过一个多路选择器将选择的行地址输出到行译码器，刷新计数器的位数也是 11 位，一次刷新相当于对一行数据进行一次读操作，每次读后需再生。

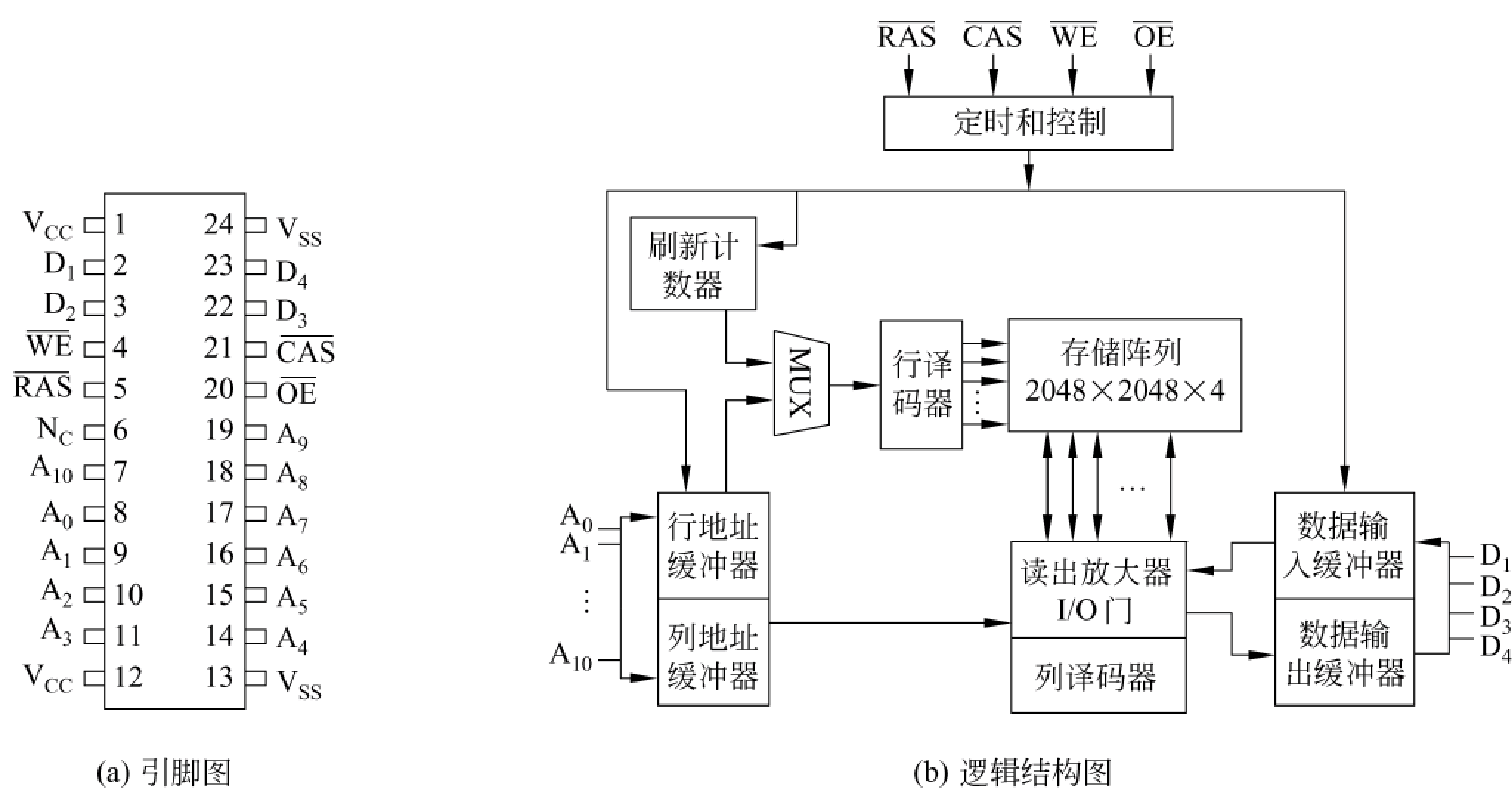
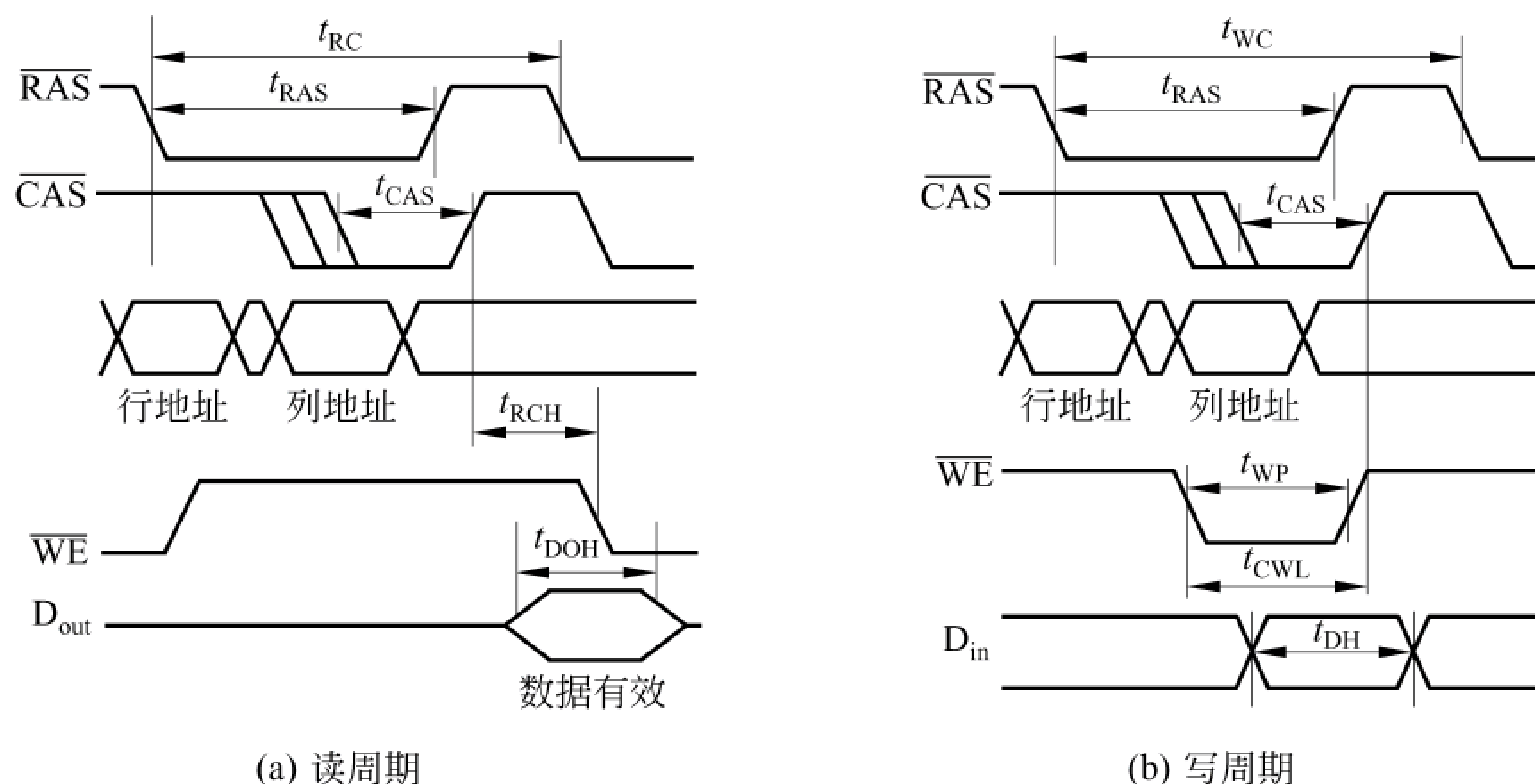


图 4.8　4M×4 位 DRAM 芯片

2. DRAM 芯片读写周期

图 4.9 是 DRAM 芯片读写周期时序示意图。



(a) 读周期　　(b) 写周期

图 4.9　DRAM 芯片读写周期时序图

读周期中，为使芯片能正确地接受行、列地址并实现读操作，各信号的时间关系应符合下面的要求。

(1) 行地址必须在$\overline{\text{RAS}}$信号有效之前送到芯片的地址输入端。

(2) $\overline{\text{CAS}}$信号应滞后$\overline{\text{RAS}}$一段时间，并滞后列地址送到芯片地址输入端的时间。

(3) $\overline{\text{RAS}}$、$\overline{\text{CAS}}$的时延分别为 t_{RAS} 和 t_{CAS}，它们应有足够的宽度。

(4) $\overline{\text{WE}}$信号为高电平，并在$\overline{\text{CAS}}$有效之前建立。

(5) 每次读后要再生，即重新写入一次。

在写周期中，$\overline{\text{RAS}}$与$\overline{\text{CAS}}$之间的关系以及与地址信息间的关系和读周期相同。但还有两点不同。

(1) $\overline{\text{WE}}$信号为低电平，并在$\overline{\text{CAS}}$信号有效之前建立。

(2) 写数据必须在$\overline{\text{CAS}}$有效之前出现在 D_{in} 端。

3. DRAM 芯片的刷新

DRAM 的存储阵列中所有存储电容必须周期性地重新充电，这一过程称为“刷新”。“刷新”可以采用“读出”的方法进行，根据读出内容对相应单元进行“重写”，即读后再生。刷新时只给各芯片送行地址和$\overline{\text{RAS}}$信号，这样芯片中一行的所有元素被选中并进行“读出”操作。因此，“刷新”操作按行进行，一次可刷新一行所有元素。对于由上述图 4.8 中芯片组成的存储器，其存储体为 2048×2048×4 结构，所以，只要 2048 次刷新操作就可将整个存储器刷新一遍。

刷新不需要外部提供行地址信息，这是一个内部的自动操作。DRAM 芯片内部有一个行地址生成器（也称刷新计数器），由它自动生成行地址。由于刷新是针对一行中所有存储元进行，所以无须进行列寻址。

刷新周期定义为从上次对整个存储器刷新结束开始到下次对整个存储器全部刷新一遍为止的时间间隔，也就是相邻两次对某个特定行进行刷新的时间间隔。应该隔多长时间重

复一次刷新呢？目前公认的标准是存储体中电容的数据有效保存期的上限 64ms，也就是说每一行的刷新周期是 64ms，但有些器件也不一定是 64ms。

有三种刷新方式：集中、分散和异步。

(1) 集中刷新。在整个刷新间隔内，前一段时间用于正常的读写操作，而在后一段时间停止读写操作，集中逐行进行刷新。因为刷新操作同时对所有芯片进行，所以刷新过程中整个存储器都不能进行正常读写操作。由于集中刷新时间较长，因此处于这种非正常工作状态的时间较长，极大影响系统执行效率，所以很少采用集中式刷新方式。

(2) 分散刷新。将一个存储周期分为两段，前一段时间用于正常读写操作，后一段时间用于刷新操作。这样，不存在死区，但是每个存储周期的时间加长。所以这种方式也很少使用。

(3) 异步刷新。结合上述两种方式，将一个刷新周期分配给所有行，使得在一个刷新周期内每一行都至少被刷新一次，且仅被刷新一次。若刷新周期为 64ms，则相邻两行之间的刷新间隔就是 64ms/行数。例如，对于规格是 4096 行、刷新周期为 64ms 的 DRAM 存储器，其相邻两行的刷新间隔为 15.625μs，8192 行时则为 7.8125μs。这种方式比前两种效率高，目前用得较多。

4. SDRAM 芯片技术

目前主存常用的是基于 SDRAM(Synchronous DRAM)芯片技术的内存条，包括 DDR SDRAM、DDR2 SDRAM 和 DDR3 SDRAM 等。

SDRAM 的工作方式与传统的 DRAM 有很大不同。传统 DRAM 与 CPU 之间采用异步方式交换数据，CPU 发出地址和控制信号后，经过一段延迟时间，数据才读出或写入。在这段时间里，CPU 不断采样 DRAM 的完成信号，在没有完成之前，CPU 插入等待状态而不能做其他工作。而 SDRAM 芯片则不同，其读写受系统时钟控制，因此与 CPU 之间采用同步方式交换数据。它将 CPU 或其他主设备发出的地址和控制信息锁存起来，经过确定的几个时钟周期后给出响应。因此，主设备在这段时间内，可以安全地进行其他操作。

SDRAM 的每一步操作都在外部系统时钟 CLK 的控制下进行，支持突发(burst)传输方式。只要在第一次存取时给出首地址，以后按地址顺序读写即可，而不再需要地址建立时间和行、列预充电时间，就能连续快速地从行缓冲器中输出一连串数据。内部的工作方式寄存器(也称模式寄存器)可用来设置传送数据的长度以及从收到读命令到开始传送数据的延迟时间等，前者称为突发长度(Burst Lengths，BL)，后者称为 CAS 潜伏期(CAS Latency，CL)。根据所设定的 BL 和 CL，CPU 可以确定何时开始从总线上取数以及连续取多少个数据。在开始的第一个数据读出后，同一行的所有数据都被送到行缓冲器中，因此，以后每个时钟可从 SDRAM 读取一个数据，并在下一个时钟内通过总线传送到 CPU。

基于 SDRAM 技术的芯片的工作过程大致如下。(1)在 CLK 时钟上升沿片选信号($\overline{CS}$)和行地址选通信号($\overline{RAS}$)有效；(2)经过一段延时 t_{RCD}(RAS to CAS Delay)，列选通信号$\overline{CAS}$有效，此时，行、列地址被确定，已选中具体的存储单元；(3)对于读操作，再经过一个 CAS 潜伏期后，输出数据开始有效，其后的每个时钟都有一个或多个数据连续从总线上传出，直到完成突发长度 BL 指定的所有数据的传送。对于写操作，则没有 CL 延时而直接开

始写入。由于只有读操作才有 CL，所以 CL 又被称为读取潜伏期(Read Latency，RL)。t_{RCD} 和 CL 都是以时钟周期 T_{CK} 为单位，例如，对于 PC100 SDRAM 来说，当 T_{CK} 为 10ns，CL 为 2 时，则 CAS 潜伏期延时为 20ns。BL 可用的选项为 1、2、4、8 等，当 BL 为 1 时，则是非突发传输方式。

DDR(Double Data Rate)SDRAM 是对标准 SDRAM 的改进设计，通过芯片内部读写缓冲数据的两位预取，并利用存储器总线上时钟信号的上升沿与下降沿进行两次传送来同步，以实现一个时钟内传送两次数据。类似的技术可以实现一个时钟内传送 4 个数据(DDR2)或 8 个数据(DDR3)。

4.3 半导体只读存储器和 Flash 存储器

4.3.1 半导体只读存储器

根据只读存储器的工艺，可分为 MROM、PROM、EPROM 和 EEPROM(E^2PROM)等类型。

1. 掩膜只读存储器(MROM)

掩膜只读存储器(Mask ROM)中存储的信息由生产厂家在掩膜工艺过程中“写入”，用户不能修改。掩膜 ROM 有双极型和 MOS 型两种。存储矩阵可以采用单译码结构，也可以采用双译码结构。在单译码结构中，当译码器“选中”某字选线时，被选中字的各位同时向各自的位线送“读出”结果。

MROM 存储内容固定，所以可靠性高，但灵活性差，生产周期长，用户和厂家间依赖性大，只适合定型批量生产。

2. 可编程只读存储器(PROM)

可编程只读存储器(Programmable ROM)芯片出厂时内容全部为 0(半成品)，用户可用专门的 PROM 写入器将信息写入，所以称为可编程型 ROM，但写入不可逆，某位写入 1 后，就不能再变为 0，因此称为一次编程型只读存储器。

PROM 有两种工艺：熔丝型和反向二极管型。熔丝型较常用，在行、列交点处连接一段熔丝，存入 0。若该位需写入 1，则让它通过较大电流，使熔丝烧断。反向二极管型在行、列交点处有一对反向的二极管，因反向而不导通，存入 0。若该位需写入 1，则在相应行、列之间加较高电压，将其中反向二极管永久性击穿，留正向可导通的一只二极管。反向二极管型也被称为 P-N 结破坏型。

3. 可擦除可编程只读存储器(EPROM)

可擦除可编程只读存储器(Erasable Programmable ROM)允许用户通过某种编程器向 ROM 芯片中写入信息，并可擦除所有信息后重新写入。可反复擦除-写入多次。一般把 EPROM 芯片上的石英窗口对着紫外线灯(12MW/cm^2 规格)，距离 3cm 远，照射 8～20 分钟，即可抹除芯片上的全部信息。

EPROM 比 MROM 和 PROM 灵活与实用，但采用 MOS 工艺，速度比较慢。擦除时，芯片中所有信息都会消失，不灵活，因而又引入了一种电可擦除的 EPROM。

4. 电擦除电改写只读存储器(EEPROM)

电擦除电改写只读存储器(Electrically Erasable Programmable ROM)又叫 EEPROM 或 E^2PROM。这种存储器在读数据的方式上与 EPROM 完全一样,但它有一个明显优点,即可用电来擦除和重编程,因此,可以选择只删除个别字,而不像 EPROM 那样,每次都要抹除芯片上的全部信息,这给现场重编程带来极大的方便。不过 E^2PROM 在每次写入操作时先执行一个自动擦除,因此比 RAM 的写操作慢得多,大约 20ms。一般 E^2ROM 可进行数千次擦除,在不受干扰时,所存放的数据至少可维持 10 年,多则达 20 多年。

4.3.2 半导体 Flash 存储器

Flash 存储器也称为闪存,是高密度非易失性读写存储器,它兼有 RAM 和 ROM 的优点,而且功耗低、集成度高,不需后备电源。

这种器件沿用了 EPROM 的简单结构和浮栅/热电子注入的编程写入方式,又兼备 E^2PROM 的可擦除特点,可在计算机内进行擦除和编程写入。因此又称为快擦型电可擦除重编程 ROM。目前被广泛使用的 U 盘和存储卡等都属于 Flash 存储器。

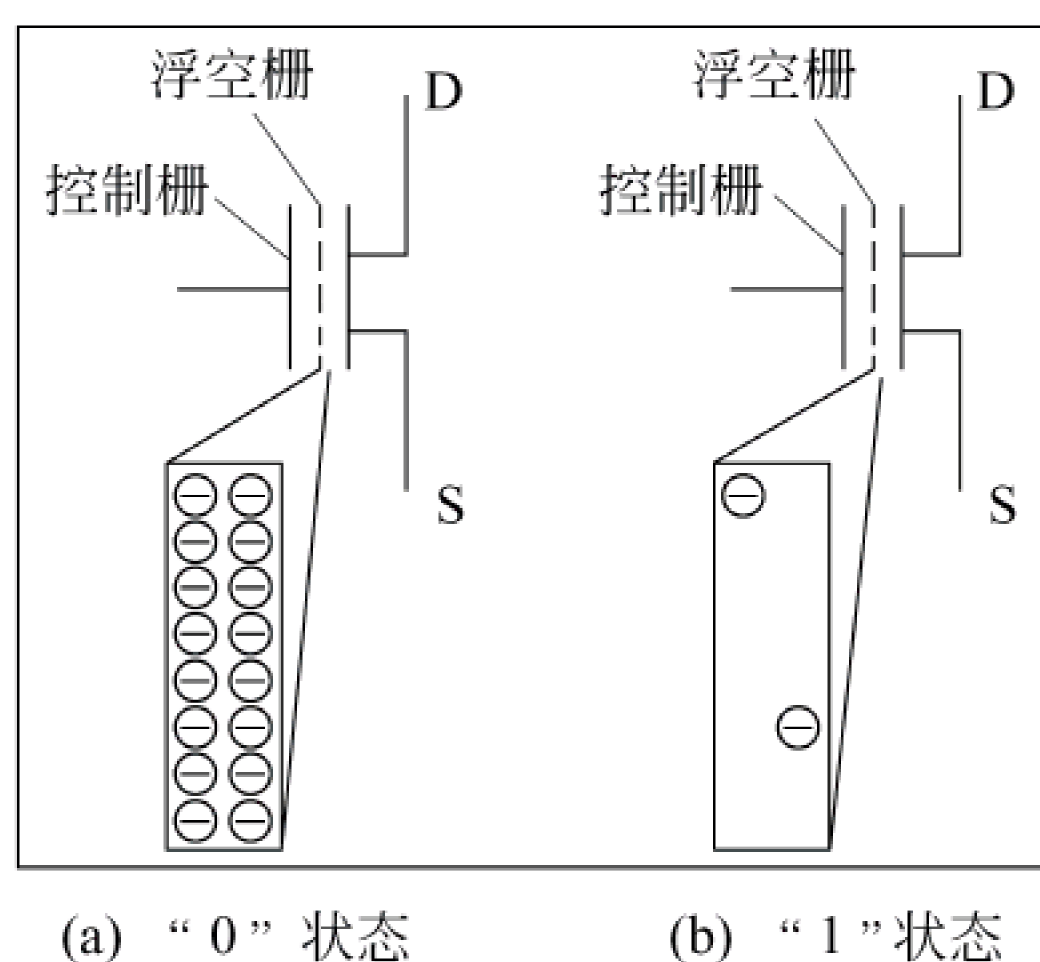


图 4.10 Flash 存储元

1. Flash 存储元

Flash 存储元是在 EPROM 存储元基础上发展起来的。如图 4.10 所示是一个 Flash 存储元,每个存储元由单个 MOS 管组成,包括漏极 D、源极 S、控制栅和浮空栅。当控制栅加上足够的正电压时,浮空栅将储存大量电子,即带有许多负电荷,可将存储元的这种状态定义为“0”;当控制栅不加正电压,则浮空栅少带或不带负电荷,将这种状态定义为“1”。

2. Flash 存储器的基本操作

闪存有三种基本操作:编程、读取、擦除。

编程操作:最初所有存储元都是“1”状态,通过“编程”,在需要改写为“0”的存储元的控制栅加上一个正电压 V_P,如图 4.11(a)所示。一旦某存储元被编程,则存储的数据可保持 100 年而无须外电源。

擦除操作:采用电擦除。即在所有存储元的源极 S 加正电压 V_E,使浮空栅中的电子被吸收掉,从而使所有存储元都变成“1”状态,如图 4.11(b)所示。因此,写的过程实际上是先全部擦除,使全都变成“1”状态后再在需要的地方改写为“0”。

读取操作:在控制栅加上正电压 V_R,若原存为“0”,则读出电路检测不到电流,如图 4.12(a)所示;若原存为“1”,则浮空栅不带负电荷,控制栅上的正电压足以开启晶体管,电源 V_d 提供从漏极 D 到源极 S 的电流,读出电路检测到电流,如图 4.12(b)所示。

从上述基本原理可以看出,Flash 存储器的读操作速度和写操作速度相差很大,其读取速度与半导体 RAM 芯片相当,而写数据(快擦—编程)的速度则与硬磁盘存储器相当。

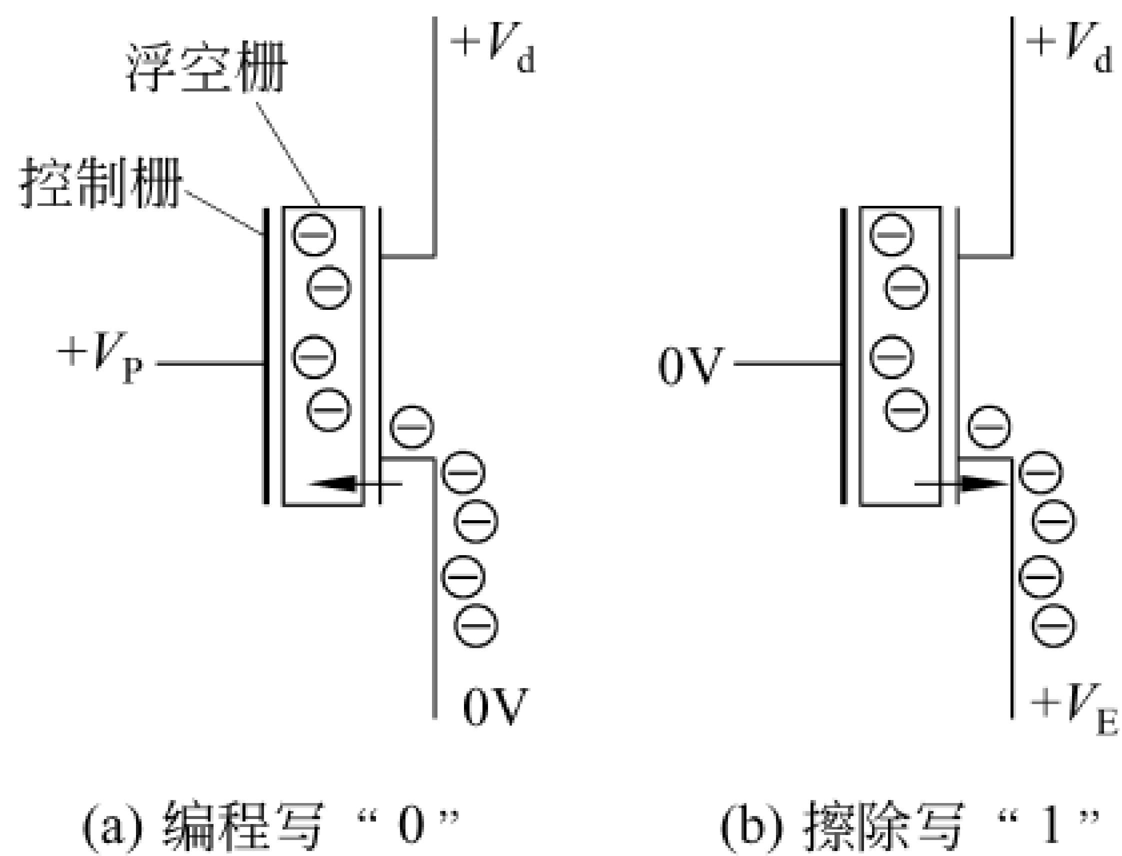


图 4.11　Flash 存储元的写入

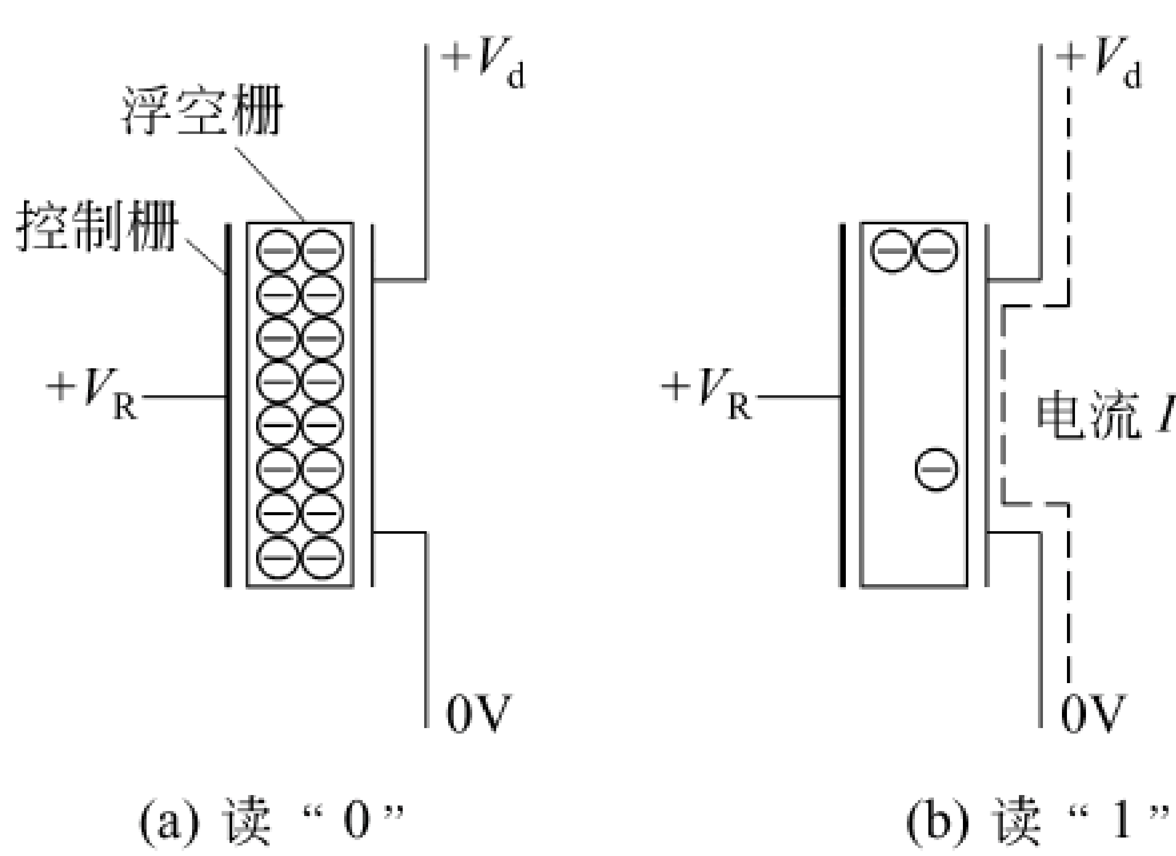
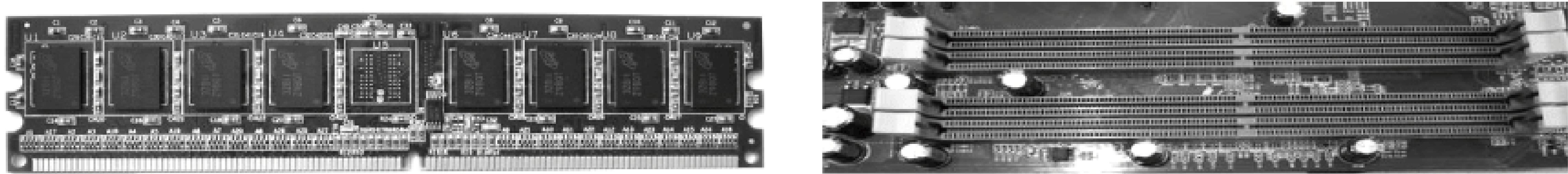


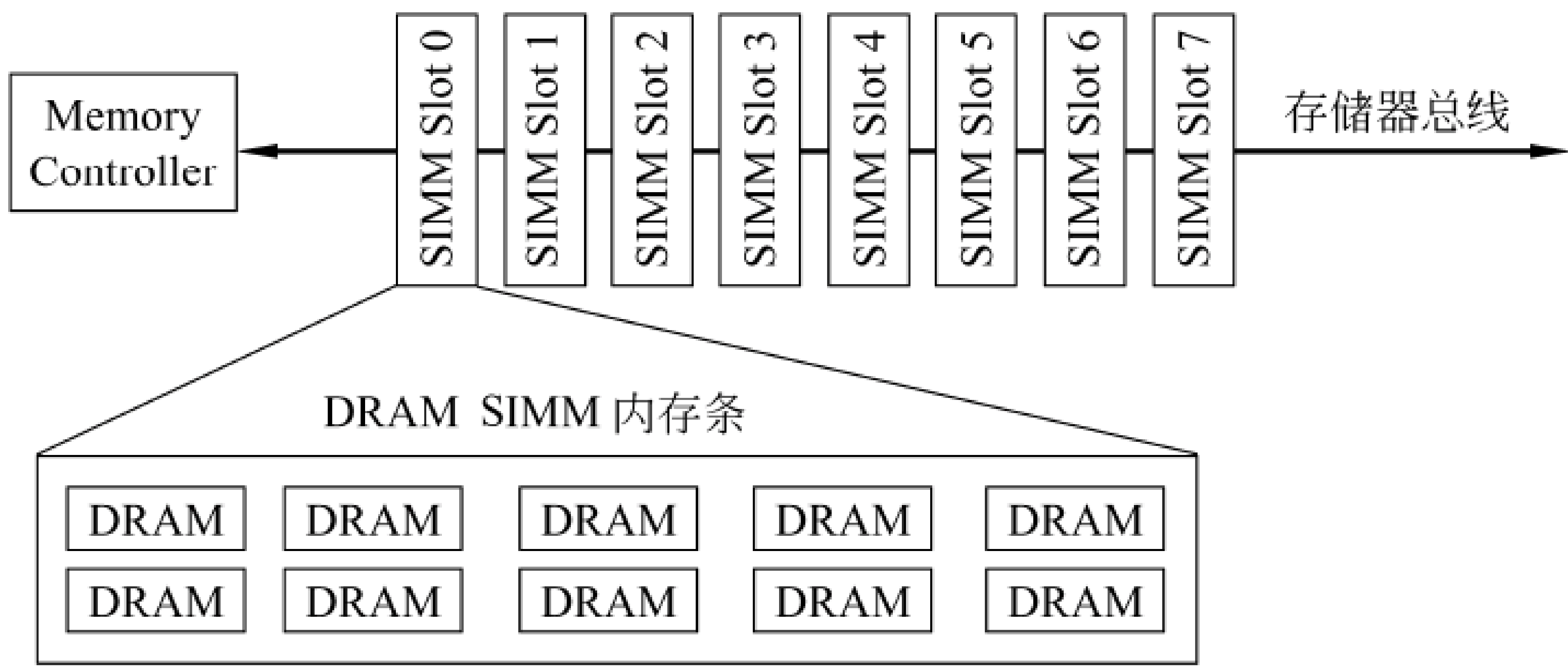
图 4.12　Flash 存储元的读出

4.4　存储器芯片的扩展及其与 CPU 的连接

受集成度和功耗等因素的限制，单个芯片的容量不可能很大，所以往往通过存储器芯片的扩展技术，将多个芯片做在一个内存模块上，然后由多个内存模块以及主板或扩充板上的 RAM 芯片和 ROM 芯片组成一台计算机所需的主存空间，再通过系统总线和 CPU 相连，如图 4.13 所示。图 4.13(a)是内存条和内存条插槽(slot)示意图，图 4.13(b)是存储控制器(Memory Controller)、存储器总线、内存条和 DRAM 芯片之间的连接关系示意图。



(a) 内存条和内存条插槽



(b) 存储控制器、存储器总线、内存条和 DRAM 芯片之间的连接

图 4.13　DRAM 芯片在系统中的位置及其连接关系

4.4.1　存储器芯片的扩展

由若干个存储器芯片构成一个存储器时，需要在字方向和位方向上进行扩展。

1. 位扩展

用若干片位数较少的存储器芯片构成给定字长的存储器时，需要进行位扩展。例如，用

8 片 4096×1 位的芯片构成 4K×8 位的存储器，需要在位方向上扩展 8 倍，而字方向上无须扩展。位扩展时，各芯片上的地址线及读写控制线对应相接，而数据线单独引出。

2. 字扩展

字扩展是容量的扩充，位数不变。例如，用 16K×8 位的存储芯片在字方向上扩展 4 倍，构成一个 64K×8 位的存储器。字扩展时，芯片的地址线、读写控制线等引脚对应相接，片选信号则分别与外部译码器的各个译码输出端相连。

3. 字、位同时扩展

当芯片在容量和位数都不满足存储器要求的情况下，需要对字和位同时扩展。例如，用 16K×4 位的存储芯片在字方向上扩展 4 倍、位方向上扩展两倍，可构成一个 64K×8 位的存储器。

如图 4.14 所示，给出了用 8 个 16M×8 位的 DRAM 芯片扩展构成一个 128MB 主存的示意图。每片 DRAM 芯片中有一个 4096×4096×8 位的存储阵列，所以，行地址和列地址各 12 位，有 8 个位平面，在每个行、列交叉处的 8 个位平面数据同时进行读写，所以一个芯片每次读写 8 位。芯片中每一行有 4096 列，每列有 8 位。选中某一行并读出到行缓冲后，再由列地址选择其中的一列（8 个二进位）送到存储控制器，组合成需要的传输宽度（如 64 位），再通过存储器总线进行传输。

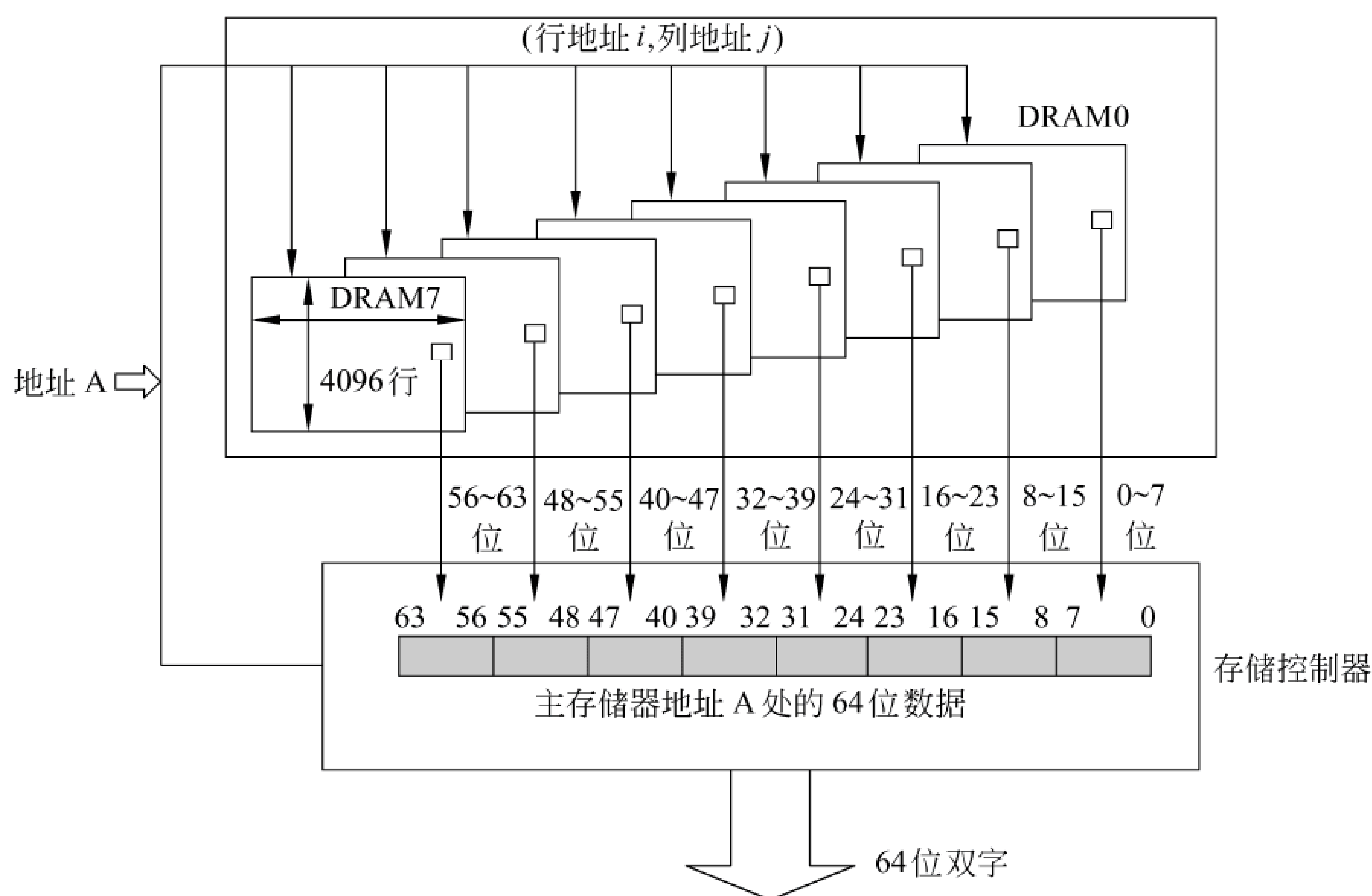


图 4.14 DRAM 芯片的扩展

4.4.2 存储器芯片与 CPU 的连接

在 CPU 与存储器芯片连接时，需考虑下述几方面的问题：(1) CPU 的负载能力：当存储器芯片较多时，需在 CPU 与存储器芯片之间增加必要的缓冲和驱动电路；(2) 速度匹配问题：存储器芯片的存取速度与 CPU 相比，有很大差距，因此，在设计 CPU 与存储器芯片

的连接线路时，要考虑速度匹配问题；(3)多片存储芯片的选通：在 CPU 和存储芯片之间需增加外部译码电路，产生片选信号，这样才能保证在不同地址范围内访问不同的芯片；(4)读写控制信号：CPU 的读写控制信号不一定与存储芯片引脚定义的控制信号相符，所以有时需要增加某些附加线路来实现正确的控制。

1. 地址线的连接

CPU 的地址线数决定了整个主存空间的寻址范围，因此，它一般比存储芯片的地址引脚线多。连续编址时，将 CPU 地址线的低位和存储芯片地址线相连，高位地址用作字扩展时的片选信号的译码；交叉编址时，则低位地址用作片选信号译码。

2. 数据线的连接

CPU 的数据线数决定了一次可读写的最大数据宽度，因此也往往比存储芯片的数据线多。通常将 CPU 数据线连到多个进行位扩展的芯片中，使扩展后的位数与 CPU 数据线数相等。

3. 控制线的连接

如果 CPU 读写命令线和存储芯片的读写控制线都是一根，并且控制电平信号一致，则可以直接相连。若 CPU 读写命令线分开，则需要分别进行连接。

CPU 中的访存信号线$\overline{MREQ}$用来确定访问的是主存还是 I/O 端口。只有要求访问主存($\overline{MREQ}$信号为低电平)时才需要选择存储芯片，所以，存储芯片的片选信号$\overline{CS}$与$\overline{MREQ}$信号有关。另外，部分地址线也与片选信号有关。

在选择存储芯片的类型和数量时，必须先确定好 ROM 区和 RAM 区的地址范围。通常将系统加电时用于系统引导的一组程序(包括加电自检程序、系统配置和设置程序、系统自举装入程序、底层设备驱动程序及底层中断服务程序等)存放在 ROM 区，因此这些区域应选用 ROM 芯片构造；操作系统内核和用户程序则存放在 RAM 区，应选用 RAM 芯片构造。

例 4.1 假定某计算机 CPU 地址线为 $A_{15}\sim A_0$，数据线为 $D_7\sim D_0$，$\overline{WR}$为读写控制信号，$\overline{MREQ}$为访存请求信号。0000H～3FFFH 为用于系统加电引导的一组程序区，4000H～FFFFH 为操作系统内核和用户程序区。现用 8K×4 位的 ROM 芯片和 16K×8 位的 RAM芯片构成该存储器，并按字节编址。要求：说明地址译码方案，并将 ROM 芯片、RAM 芯片与 CPU 连接。

解：CPU 地址线为 $A_{15}\sim A_0$，数据线为 $D_7\sim D_0$，因此，最大存储容量为 64KB。因为按字节编址，所以地址空间为 0000H～FFFFH。因此，题目要求对所有 64KB 空间进行完全配置。

根据题意，知 0000H～3FFFH 为 ROM 区，因此，ROM 区的高两位总是 00，对低 14 位进行全译码。因为 ROM 区大小为 $2^{14}\times 8$ 位＝16K×8 位＝16KB，所以 ROM 芯片数为 16K×8 位/8K×4 位＝2×2＝4，因而字方向扩展两倍，位方向扩展两倍。ROM 芯片内地址位数为 13 位，连到 CPU 低 13 位地址线 $A_{12}\sim A_0$。

根据题意，知 4000H～FFFFH 为 RAM 区，因此，RAM 区的高两位是 01、10、11，低 14 位为全译码。因为 RAM 区大小为 $3\times 2^{14}\times 8$ 位＝3×16K×8 位＝48KB，RAM 芯片数

为48K×8位/16K×8位=3×1=3,因而只在字方向上扩展3倍,位方向上不扩展。RAM芯片内地址位数为14位,连到CPU低14位地址线A_{13}~A_0。

ROM芯片、RAM芯片和CPU的连接如图4.15所示。

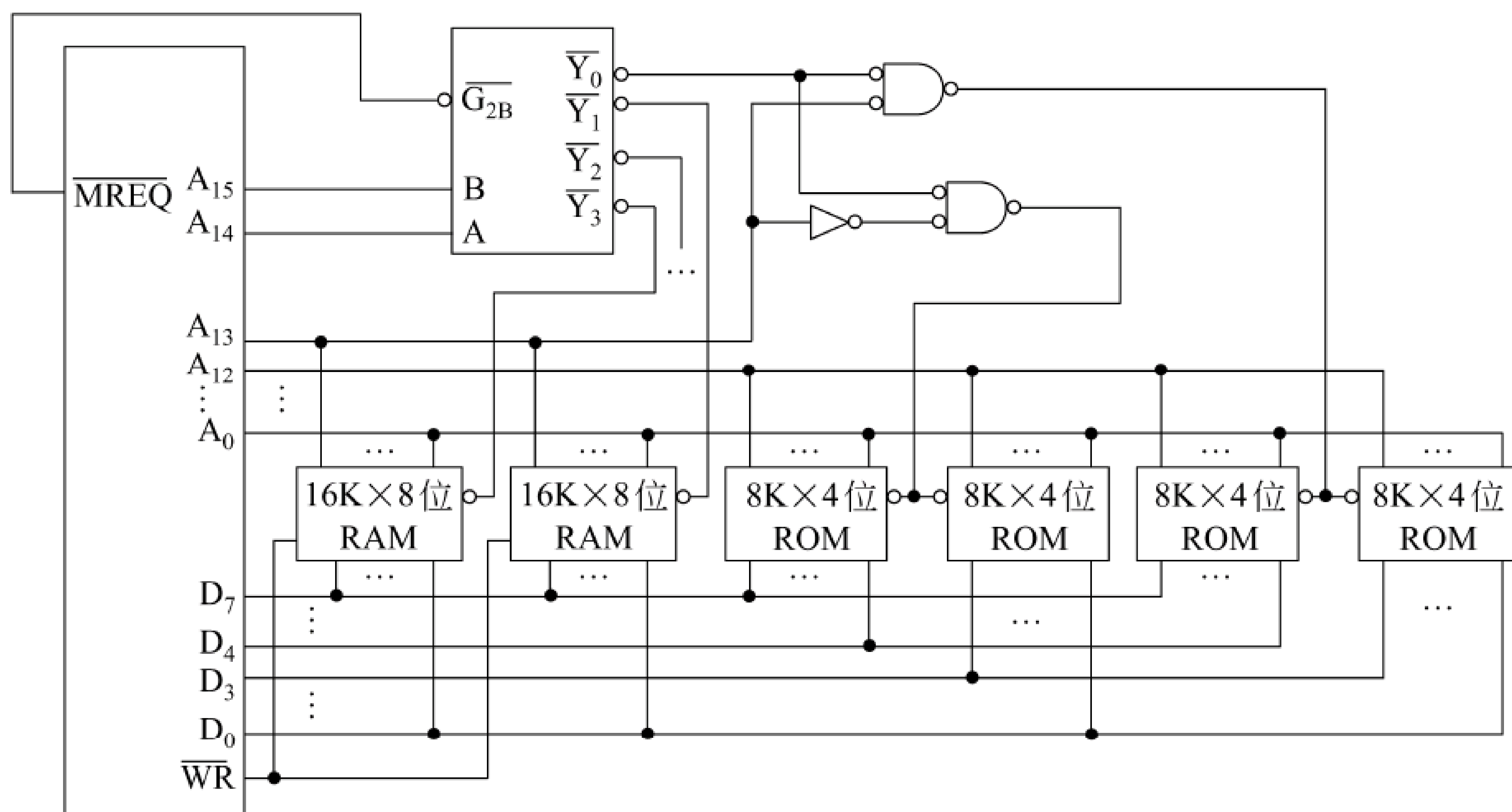


图4.15 例4.1中ROM芯片、RAM芯片和CPU的连接

4.5 并行存储器结构技术

由于CPU和主存所使用的半导体器件工艺不同,两者速度上的差距导致快速的CPU等待慢速的主存储器,为此需要想办法提高主存的速度。

改善主存速度的方法可从三个方面考虑。

(1) 提高DRAM芯片本身的速度,如前述的各种基于SDRAM的技术。

(2) 采用并行结构技术,本节介绍的双口存储器和多模块存储器就是利用时间并行和空间并行性在结构上进行优化的技术。

(3) 在CPU和主存之间增加高速缓冲存储器。这是第4.6节将要介绍的内容。

4.5.1 双口存储器

双口存储器在一个存储器中提供两组独立的读写控制电路和两个读写端口,因而可以同时提供两个数据的并行读写,是一种空间并行技术。

如图4.16所示是一个双口存储器逻辑结构示意图。每个读写口都有一套独立的地址缓存器和译码电路,两套电路并行独立工作。当A、B两个端口地址不相同时,可以向存储体一次读写两个单元的内容;当A、B两个端口地址相同时,发生冲突,可按照特定的优先顺序选择其中一个端口进行读写。

通常用双口RAM作为通用寄存器组GRS或指令预取部件,也有一些计算机把双口RAM设计成一个端口面向CPU,另一个端口面向输入输出(如I/O处理器或DMA设备)。也可以用在多机系统中,实现双口或多口存储器与多CPU之间的信息交换。

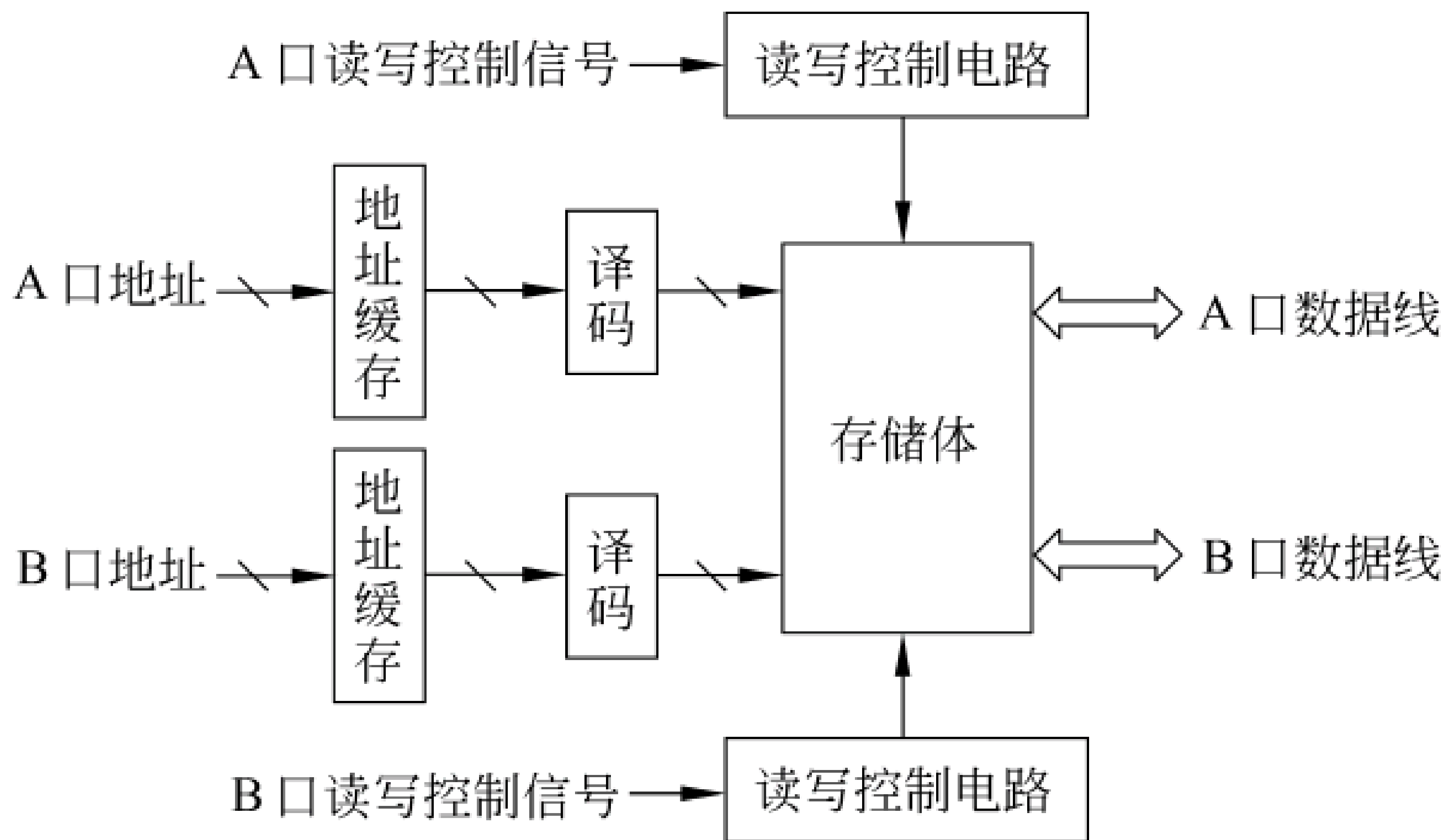


图 4.16　双口存储器逻辑结构

4.5.2　多模块存储器

多模块存储器是一种空间并行技术，利用多个结构完全相同的存储模块的并行工作来增加存储器的吞吐率。每个存储模块具有相同的容量和存取速度，并各自具有独立的地址寄存器 MAR、数据寄存器 MDR、地址译码器、驱动电路和读写电路，因此，它们能独立并行工作。

根据不同的编址方式，多模块存储器分为连续编址和交叉编址两种结构。

1. 连续编址方式

连续编址的多模块主存储器中，主存地址的高位表示模块号(或体号)，低位表示模块内地址(或体内地址)，因此，也称为按高位地址划分方式，地址在模块内连续。图 4.17 是连续编址方式示意图，存储器共有 4 个模块 M_0～M_3，每个模块有 n 个单元，M_0 的地址范围为 0～$n-1$，…，M_3 的地址范围为 $3n$～$4n-1$。连续编址方式下，总是把低位的体内地址送到由高位体号所确定的模块内进行译码。

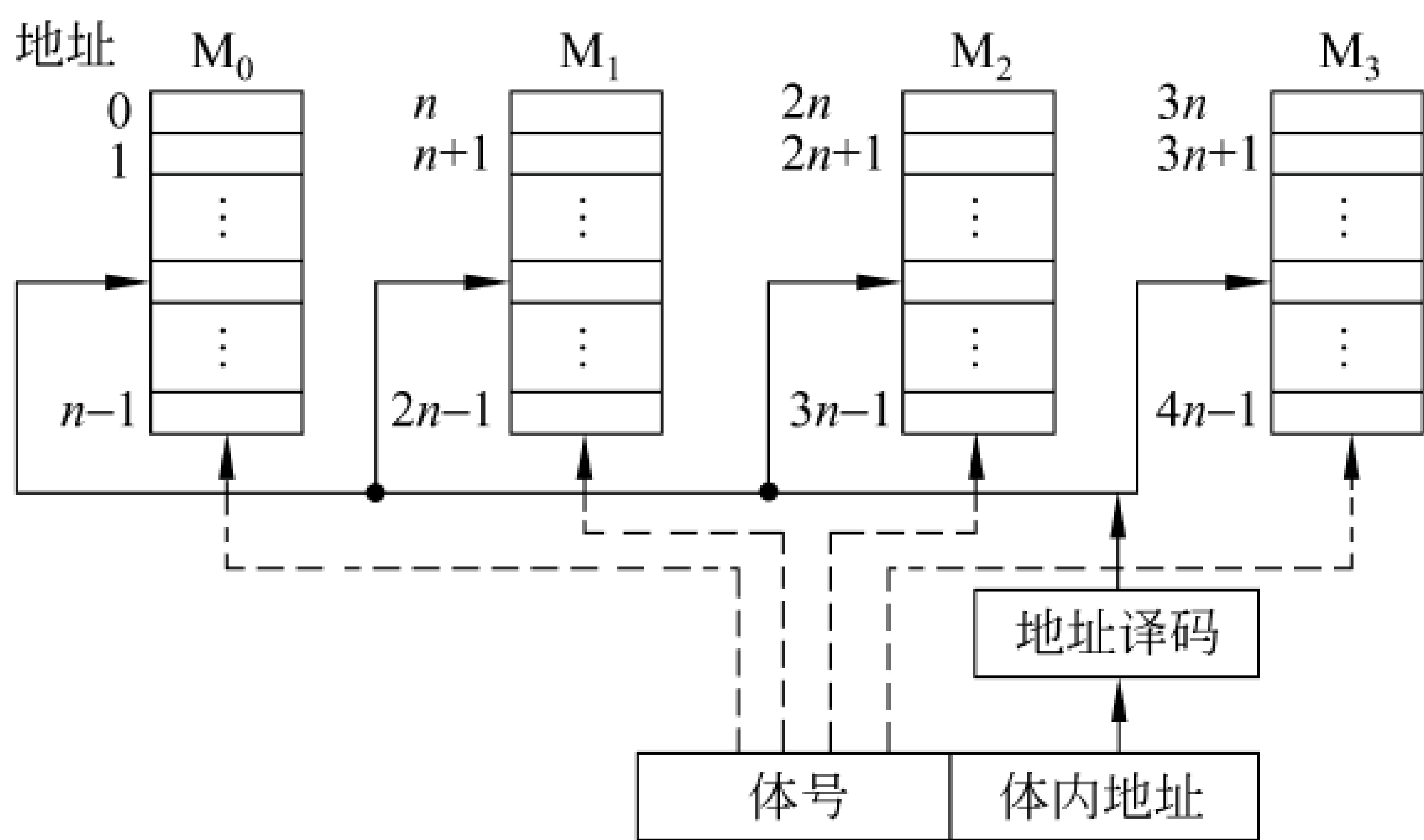


图 4.17　连续编址的多模块存储器

对于连续编址的多模块主存储器，当访问一个连续主存块时，总是先在一个模块内访问，等到该模块全部单元访问完才转到下一个模块访问。现代计算机采用层次化存储体系结构，在 CPU 和主存之间加入了高速缓存 cache，因而 CPU 总是先访问 cache，在访问 cache 失效时，再去访问主存，将一个主存块内连续单元中的信息取到 cache。由此可见，cache 缺

失情况下，CPU大多只访问单个存储模块，因而不能提高存储器的吞吐率。但是，如果在CPU访问主存的同时，DMA控制器正控制在外设和另一个存储模块之间实现数据传输操作，那么，两个存储模块可独立并行工作，此时，连续编址方式能提高存储器的吞吐率。

2. 交叉编址方式

交叉编址存储器中，主存地址的低位表示模块号，高位表示模块内地址，因此，也称按低位地址划分方式。每个模块按“模 m”交叉方式编址。假定有 m 个模块体，每个体有 k 个单元，则第 $0,m,\cdots,(k-1)m$ 单元位于第0模块；第 $1,m+1,\cdots,(k-1)m+1$ 单元位于第1模块；…；第 $m-1,2m-1,\cdots km-1$ 单元位于 $m-1$ 号模块。一般模块数 m 取2的方幂，这样硬件电路比较简单。但有的机器为了减少存储器冲突，采用质数个模块，如我国银河机就取 m 为31，其硬件实现比较复杂。图4.18是模4交叉编址方式存储器示意图。交叉编址方式下，总是把高位的体内地址送到由低位体号所确定的模块内进行译码。

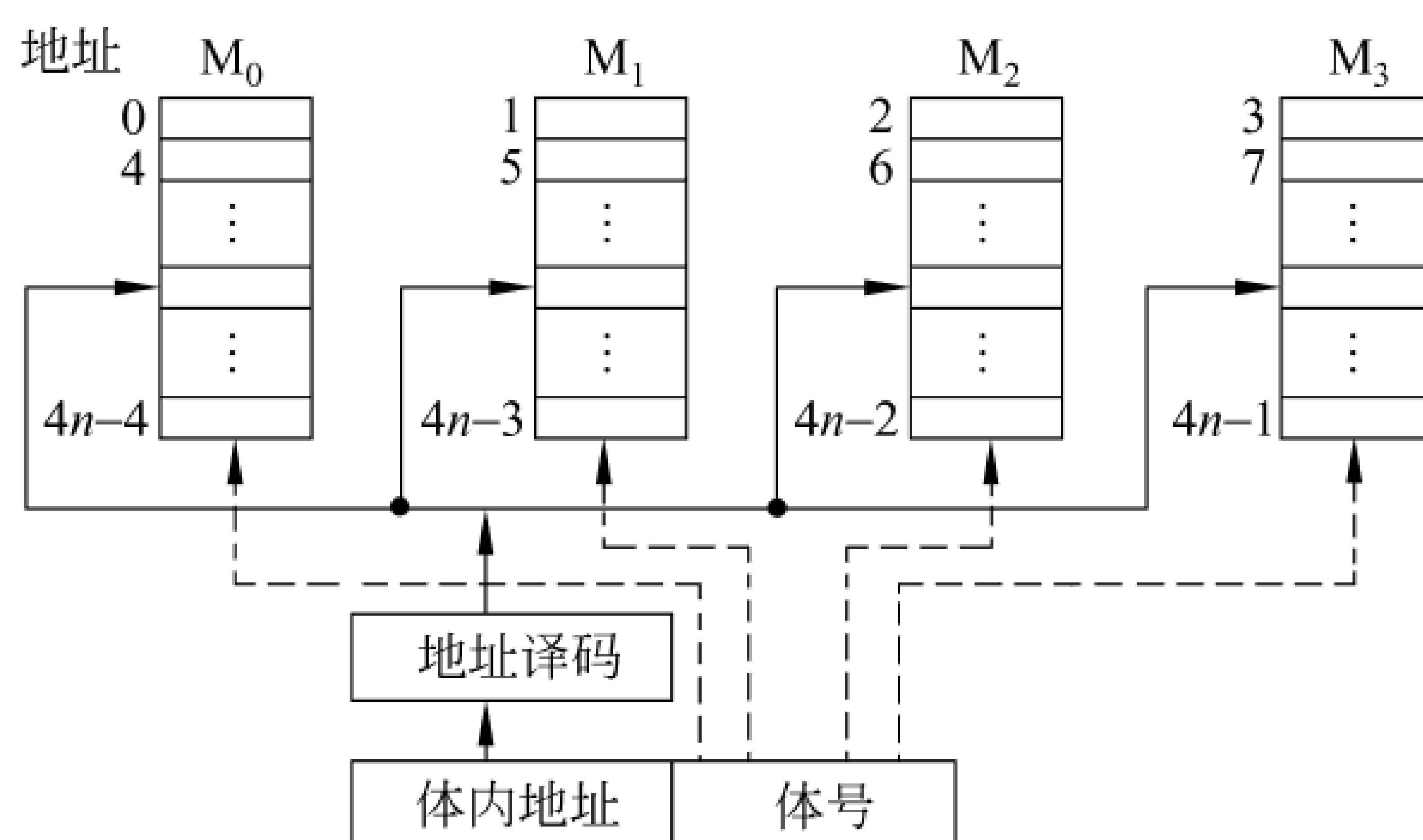


图4.18 交叉编址的多模块存储器

交叉编址多模块存储器大多采用顺序轮流启动方式。例如，若每隔 $1/m$ 个存储周期（T_M）启动一个体，则每隔 $1/m$ 个存储周期就可读出或写入一个数据，存取速度提高 m 倍。图4.19示意了这种4体交叉访问的时间关系，负脉冲为启动每个体的信号。从图4.19可看出，每个体的存储周期并没有缩短，但由于交叉访问各体，所以在一个存储周期内向CPU提供了4个存储字。若每个模块的存储字为32位，则在一个存储周期内向CPU提供了 $32\times4=128$ 位信息，因而大大提高了存储器的带宽。

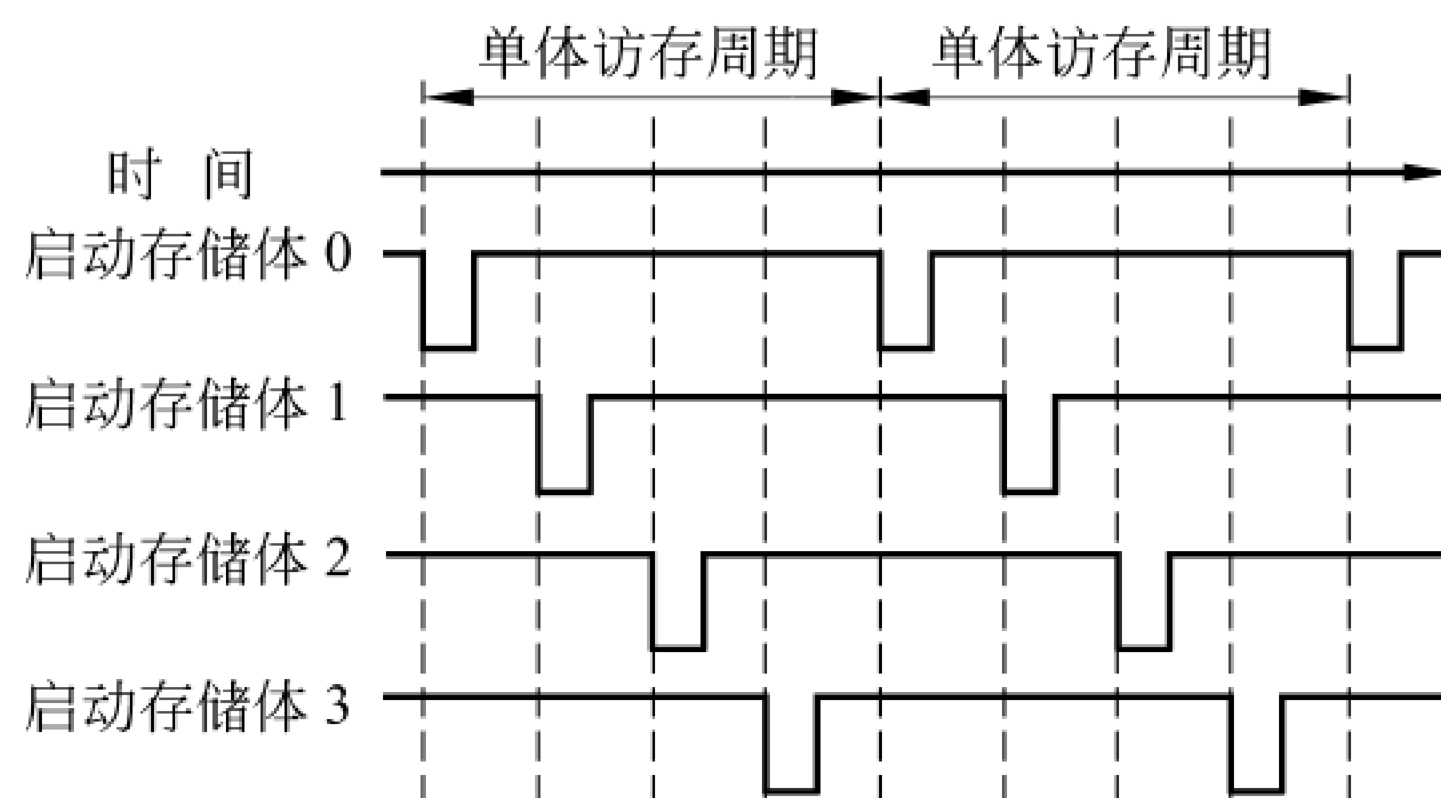


图4.19 四体交叉轮流访问方式

前面提到，现代计算机中由于在 CPU 和主存之间设置有 cache，因而通常 CPU 是对一个主存块中的连续单元进行访问，DMA 传送也是将一块连续主存单元与高速设备进行数据交换，因此，对主存的访问通常都是成块连续的，因而在交叉编址多模块存储器中访问时，可同时存取多个存储模块，因而可提高访存速度。

4.6 高速缓冲存储器

在本章 4.5 节中提到，通过提高存储芯片本身的速度或采用并行存储器结构可以缓解 CPU 和主存之间的速度匹配问题。除了这两种方法以外，在 CPU 和主存之间设置高速缓存(cache)也可以提高 CPU 访问指令和数据的速度。

4.6.1 程序访问的局部性

对大量典型程序运行情况分析的结果表明，在较短时间间隔内，程序产生的地址往往集中在存储器的一个很小范围，这种现象称为程序访问的局部性，可细分为时间局部性和空间局部性。时间局部性是指被访问的某个存储单元在一个较短的时间间隔内很可能又被访问。空间局部性是指被访问的某个存储单元的邻近单元在一个较短的时间间隔内很可能也被访问。

出现程序访问的局部性特征的原因不难理解。因为程序是由指令和数据组成的。指令在主存按序存放，地址连续，循环程序段或子程序段通常被重复执行，因此，指令具有明显的访问局部化特性；而数据在主存一般也是连续存放，特别是数组元素，常常被按序重复访问，因此，数据也具有明显的访问局部化特征。

例如，以下是一个高级语言程序段。

```
1  sum=0;
2  for (i=0; i<n; i++)
3      sum+=a[i];
4  * v=sum;
```

上述程序段对应的汇编语言程序段由 10 条指令组成，用中间语言描述如下：

```
I0           sum←0
I1           ap←A                     ;A是数组的起始地址
I2           i←0
I3           if(i>=n) goto done
I4  loop:    t← (ap)                  ;数组元素 a[i]的值
I5           sum←sum+t                ;累加值在 sum 中
I6           ap←ap+4                  ;计算下一个数组元素的地址
I7           i←i+1
I8           if (i<n) goto loop
I9  done:    V←sum                    ;累加结果保存至地址 V
```

上述描述中的 sum，ap，i，n，t 均为通用寄存器，A 和 V 为主存地址。假定每条指令占

4 个字节，每个数组元素占 4 个字节，按字节编址，则指令和数组元素在主存中的存放情况如图 4.20 所示。

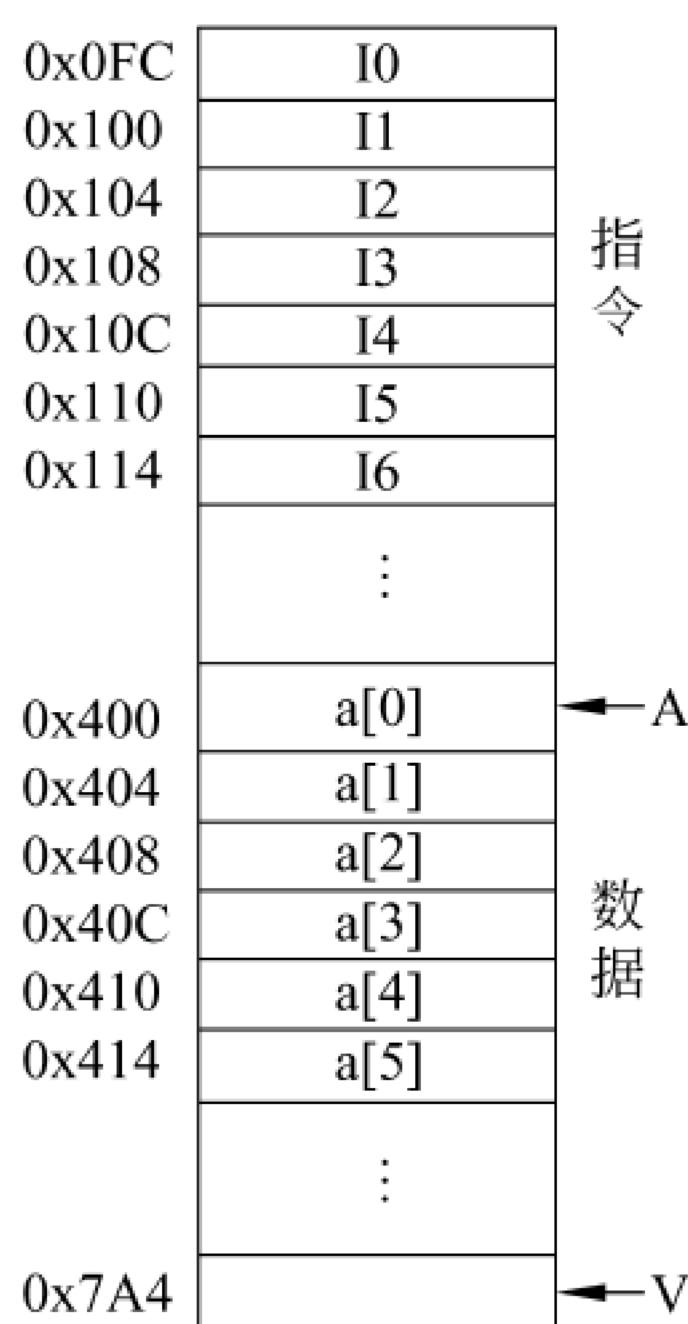


图 4.20 指令和数组在主存存放

从图 4.20 可看出，在程序执行过程中，首先指令按 I0～I3 的顺序执行，然后，指令 I4～I8 按顺序被循环执行 n 次。只要 n 足够大，程序在一段时间内，就一直在该局部区域内执行。对于取指令来说，程序对主存的访问过程为 0x0FC(I0)→0x108(I3)→0x10C(I4)→0x11C(I8)→0x120(I9)。（0x10C(I4)→0x11C(I8) 循环 n 次）

上述程序对数组的访问在指令 I4 中进行，数组下标每次加 4，按每次 4 个字节连续访问主存。因为数组在主存连续存放，因此，该程序对数据的访问过程是 0x400→0x404→0x408→0x40C→…→0x7A4。由此可见，在一段时间内，访问的数据也在局部的连续区域内。

为了更好地利用程序访问的空间局部性，通常把当前访问单元以及邻近单元作为一个主存块一起调入 cache。这个主存块的大小以及程序对数组元素的访问顺序等都对程序的性能有一定的影响。

例 4.2 假定数组元素按行优先方式存放在主存，则以下两段伪代码程序 A 和 B 中，(1)对于数组 a 的访问，哪一个空间局部性更好？哪一个时间局部性更好？(2)变量 sum 的空间局部性和时间局部性各如何？(3)对于指令访问来说，for 循环体的空间局部性和时间局部性如何？

程序段 A

```
1  int sum-array-rows (int a[M][N])
2  {
3      int i, j, sum=0;
4      for (i=0; i<M; i++)
5          for (j=0; j<N; j++)
6              sum+=a[i][j];
7      return sum;
8  }
```

程序段 B

```
1  int sum-array-cols (int a[M][N])
2  {
3      int i, j, sum=0;
4      for (j=0; j<N; j++)
5          for (i=0; i<M; i++)
6              sum+=a[i][j];
7      return sum;
8  }
```

解：假定按字节编址，M、N 都为 2048，每个数组元素占 4 个字节则指令和数据在主存的存放情况如图 4.21 所示。

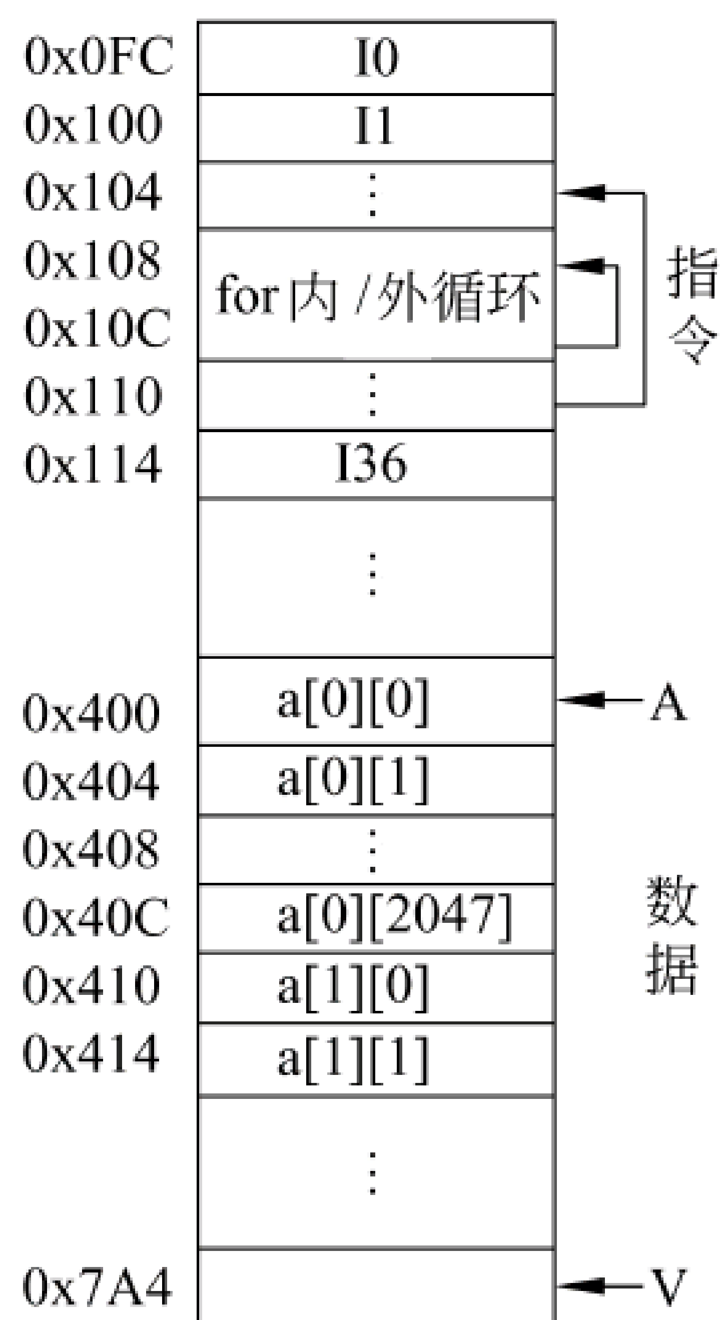


图 4.21　指令和二维数组在主存的存放

(1) 对于数组 a，程序段 A 和 B 的空间局部性相差较大。

程序 A 对数组 a 的访问顺序为 a[0][0]，a[0][1]，…，a[0][2047]；a[1][0]，a[1][1]，…，a[1][2047]；…。由此可见，访问顺序与存放顺序是一致的，故空间局部性好。

程序 B 对数组 a 的访问顺序为 a[0][0]，a[1][0]，…，a[2047][0]；a[0][1]，a[1][1]，…，a[2047][1]；…。由此可见，访问顺序与存放顺序不一致，每次访问都要跳过 2048 个数组元素，即 8192 个单元，若主存与 cache 的交换单位小于 8KB，则每次装入一个主存块到 cache 时，下个要访问的数组元素总不能被装入 cache，因而没有空间局部性。

时间局部性在程序 A 和 B 中都差，因为每个数组元素都只被访问一次。

(2) 对于变量 sum，在程序段 A 和 B 中的访问局部性是一样的。空间局部性对单个变量来说没有意义；而时间局部性在 A 和 B 中都较好，因为 sum 变量在 A 和 B 的每次循环中都要被访问。不过，通常编译器都将其分配在寄存器中，循环执行时只要取寄存器内容进行运算，最后再把寄存器的值写回到存储单元中。

(3) 对于 for 循环体，程序段 A 和 B 中的访问局部性是一样的。因为循环体内指令按序连续存放，所以空间局部性好；内循环体被连续重复执行 2048×2048 次，因此时间局部性也好。

从上述分析可以看出，虽然程序 A 和 B 的功能相同，但因为内、外两重循环的顺序不同而导致两者对数组 a 访问的空间局部性相差较大，从而带来执行时间的不同。曾有人将这两个程序(M=N=2048)放在 2GHz Pentium 4 上执行以进行比较，其实际运行结果为：程序 A 的执行只需要 59 393 288 个时钟周期，而程序 B 则需要 1 277 877 876 个时钟周期。程序 A 比程序 B 快 21.5 倍。

4.6.2　cache 的基本工作原理

cache 是一种小容量高速缓冲存储器，由快速的 SRAM 组成，直接制作在 CPU 芯片内，速度几乎与 CPU 一样快。在 CPU 和主存之间设置 cache，总是把主存中被频繁访问的活跃程序块和数据块复制到 cache 中。由于程序访问的局部性，大多数情况下，CPU 能直接从 cache 中取得指令和数据，而不必访问主存。

为便于 cache 和主存间交换信息，cache 和主存空间都被划分为相等的区域。主存中的区域称为块(block)，也称为主存块，它是 cache 和主存之间的信息交换单位；cache 中存放一个主存块的区域称为行(line)或槽(slot)。

1. cache 的有效位

在系统启动或复位时，每个 cache 行都为空，其中的信息无效，只有装入了主存块后信息才有效。为了说明 cache 行中的信息是否有效，每个 cache 行需要一个"有效位(valid bit)"。

有了有效位，就可通过将有效位清0来淘汰某cache行中的主存块，称为冲刷(flush)，装入一个新主存块时，再使有效位置1。

2. CPU在cache中的访问过程

CPU执行程序过程中，需要从主存取指令或读数据时，先检查cache中有没有要访问的信息，若有，就直接从cache中读取，而不用访问主存储器；若没有，再从主存中把当前访问信息所在的一个主存块复制到cache中，因此，cache中的内容是主存中部分内容的映像。图4.22给出了带cache的CPU执行一次访存操作的过程。

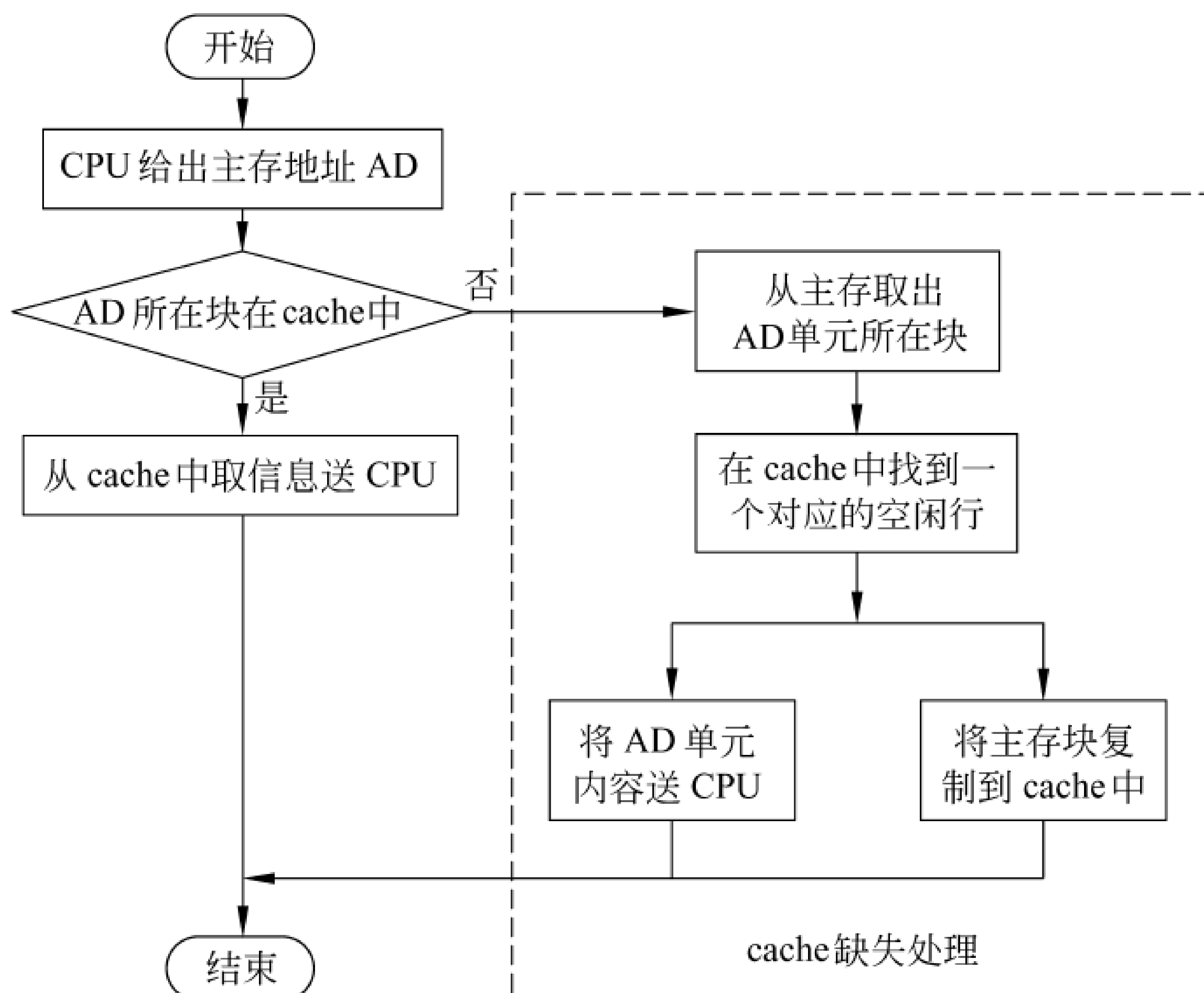


图4.22　带cache的CPU的访存操作过程

如图4.22所示，整个访存过程包括：判断信息是否在cache、从cache取信息或从主存取一个主存块到cache等工作，甚至在对应cache行已满的情况下还要替换cache中的信息。这些工作要求在一条指令执行过程中完成，因而只能由硬件来实现。因此，cache对程序员来说是透明的，程序员编程时不用考虑信息存放在主存还是在cache。

3. cache-主存层次的平均访问时间

如图4.22所示，在访存过程中，需要判断所访问信息是否在cache中。若CPU访问单元所在的块在cache中，则称cache命中(Hit)，命中的概率称为命中率p(Hit Rate)，它等于命中次数与访问总次数之比；若不在cache中，则为不命中(Miss)[①]，其概率称为缺失率(Miss Rate)，它等于不命中次数与访问总次数之比。命中时，CPU在cache中直接存取信息，所用的时间开销就是cache访问时间T_c，称为命中时间(Hit Time)；缺失时，需要从主存读取一个主存块送cache，并同时将所需信息送CPU，因此，所用时间开销为主存访问时间T_m和cache访问时间T_c，通常把从主存读入一个主存块到cache的时间T_m称为缺失损失(Miss Penalty)。

① 注：国内教材对“不命中”的说法有多种，如“失效”，“失靶”、“缺失”等，其含义一样，本教材使用“缺失”一词。

CPU 在 cache－主存层次的平均访问时间 T_a 为：

$$T_a = p\times T_c + (1-p)\times(T_m+T_c) = T_c + (1-p)\times T_m$$

由于程序访问的局部性特点，cache 的命中率可以达到很高，接近于 1。因此，虽然 Miss Penalty>>Hit Time，但最终的平均访问时间仍可接近 cache 的访问时间。

例 4.3 假定处理器时钟周期为 2ns，某程序有 1000 条指令组成，每条指令执行一次，其中的 4 条指令在取指令时，没有在 cache 中找到，其余指令都能在 cache 中取到。在执行指令过程中，该程序需要 3000 次主存数据访问，其中，6 次没有在 cache 中找到。试回答以下问题。

(1) 执行该程序得到的 cache 命中率是多少？

(2) 若 cache 中存取一个信息的时间为一个时钟周期，缺失损失为 4 个时钟周期，则 CPU 在 cache－主存层次的平均访问时间为多少？

解：(1) 执行该程序时的总访问次数为 1000＋3000＝4000，未命中次数为 4＋6＝10。cache 命中率为(4000－10)/4000＝99.75%。

(2) cache－主存层次的平均访问时间为 1＋(1－99.75%)×4＝1.01 个时钟周期，相当于 1.01×2ns＝2.02ns，与 cache 的访问时间相近。

4.6.3 cache 行和主存块之间的映射方式

cache 行中的信息取自主存中的某个块。在将主存块复制到 cache 行时，主存块和 cache 行之间必须遵循一定的映射规则，这样，CPU 要访问某个主存单元时，可以依据映射规则，到 cache 对应的行中查找要访问的信息，而不用在整个 cache 中查找。

根据不同的映射规则，主存块和 cache 行之间有以下三种映射方式。

(1) 直接(Direct)映射。每个主存块映射到 cache 的固定行中。

(2) 全相联(Full Associate)映射。每个主存块映射到 cache 的任意行中。

(3) 组相联(Set Associate)映射。每个主存块映射到 cache 的固定组的任意行中。

以下分别介绍三种映射方式。

1. 直接映射

直接映射的基本思想是把主存的每一块映射到固定的一个 cache 行中，也称模映射，其映射关系为：

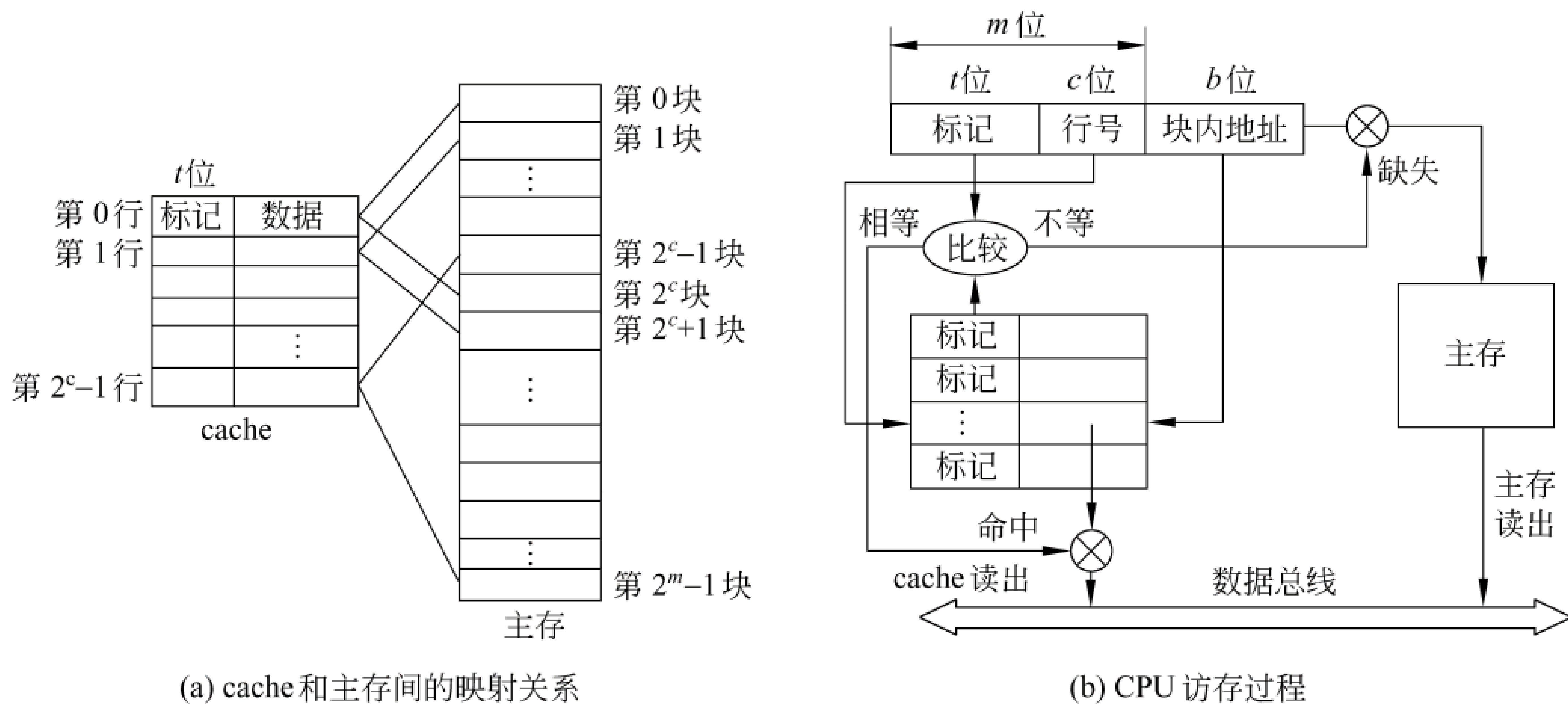
cache 行号＝主存块号 mod cache 行数

例如，假定 cache 共有 16 行，根据 100 mod 16＝4，可知：主存第 100 块应映射到 cache 的第 4 行中。

通常 cache 的行数是 2 的幂次，假定 cache 有 2^c 行，主存有 2^m 块，这个映射函数的直观含义很简单，即以 m 位主存块号中后 c 位作为对应的 cache 行号来进行 cache 映射。也就是说，m 位块号中低 c 位相同的那些内存块，即“同余”内存块，将被映射到同一个 cache 行，形成一个“多对一”的映射关系。

直接映射函数可写为 $i=j \bmod 2^c$，其中 i 是 cache 行号，j 是主存块号，c 为 cache 行号的位数，2^c 为 cache 总的行数。假定 m 为主存块号的位数，则 2^m 为主存的总块数。根据映射函数可知，主存第 0、2^c、2^{c+1}、…块映射到 cache 第 0 行；主存第 1、2^c+1、$2^{c+1}+1$、…块映射到 cache 第 1 行；…；主存第 2^c-1、$2^{c+1}-1$、$2^{c+2}-1$、…、2^m-1 块映射到 cache 第 2^c-1

行，如图 4.23(a)所示。简言之，主存块以 2^c 为模映射到 cache 的固定行中。由映射函数可看出，主存块号的低 c 位正好是它要装入的 cache 行号。在 cache 中，给每一个行设置一个 t 位长的标记(tag)，此处 $t=m-c$，主存某块调入 cache 后，就将其块号的高 t 位设置在对应 cache 行的标记中。



(a) cache和主存间的映射关系　　(b) CPU 访存过程

图 4.23　cache 和主存之间的直接映射方式

根据以上分析可知，主存地址被分成以下三个字段：

标记	cache 行号	块内地址

其中，高 t 位为标记，中间 c 位为 cache 行号(也称行索引)，剩下的低位地址为块内地址。CPU 访存过程如图 4.23(b)所示。

访存过程如下：首先根据访存地址中间的 c 位，直接找到对应的 cache 行，将对应 cache 行中的标记和主存地址的高 t 位标记进行比较，若相等并有效位为 1，则访问 cache“命中”，此时，根据主存地址中低位的块内地址，在对应的 cache 行中存取信息；若不相等或有效位为 0，则“不命中”(缺失)，此时，CPU 从主存中读出该地址所在的一块信息送到对应的 cache 行中，将有效位置 1，并将地址中的高 t 位设置为标记，同时将该地址中的内容送 CPU。

CPU 访存时，读操作和写操作的过程有一些不同，相对来说，读操作比写操作简单。因为 cache 行中的信息是主存某块的副本，所以，在写操作时会出现 cache 行和主存块数据的一致性问题。这将在第 4.5.5 节中详细介绍。

下面通过一些例子来说明 cache 设计中的一些问题。

例 4.4　假定主存和 cache 之间采用直接映射方式，块大小为 512 字。cache 容量(指数据区)为 8K 字，主存地址空间为 1M 字。问：主存地址如何划分？用图表示主存块和 cache 行之间的映射关系，假定 cache 当前为空，说明 CPU 对主存单元 0240CH 的访问过程。

解：cache 容量为 8K 字＝ 2^{13}字＝2^4 行×512 字/行＝16 行×512 字/行。

因为主存每 16 块和 cache 的 16 行一一对应，所以可将主存每 16 块看成一个块群。主存地址空间为 1M 字＝2^{20}字＝2^{11}块×512 字/块＝2^7 块群×2^4 块/块群×512 字/块。所以，主存地址位数 $n=20$，标记位数 $t=7$，行号位数 $c=4$，块内地址位数为 9。

主存地址划分以及主存块和cache行的对应关系如图4.24所示。

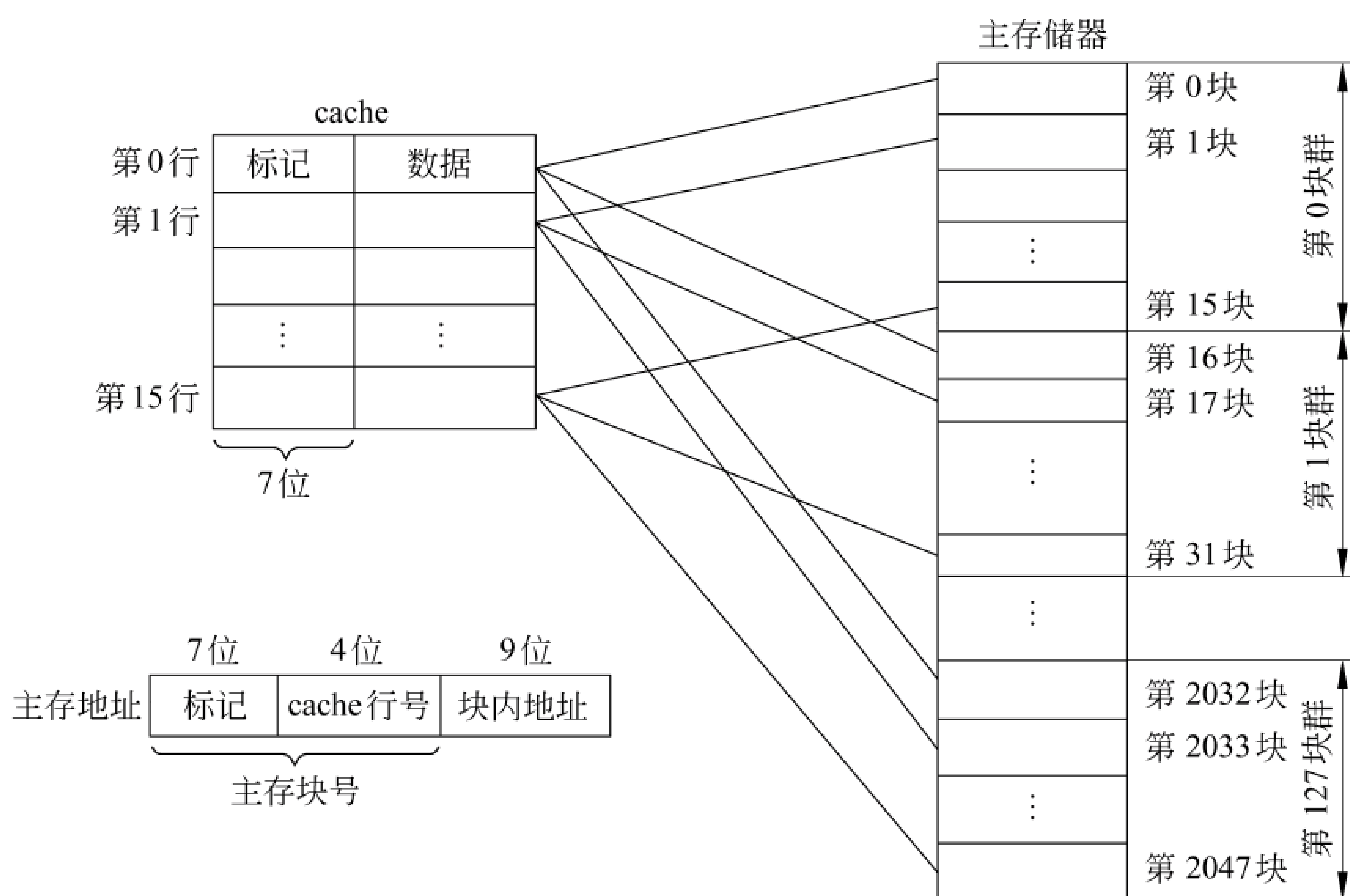


图4.24 直接映射方式下主存块和cache行对应关系

主存地址0240CH展开为二进制数为0000 0010 0100 0000 1100，所以主存地址划分为：

0000 001	0010	0 0000 1100

假定cache为空，访问0240CH单元的过程为：首先根据地址中间4位0010，找到cache第2行，因为cache为空，所以，每个cache行的有效位都为0，因此，不管第2行的标记是否等于0000 001，都不命中。此时，需要将0240CH单元所在的主存第0000 001 0010块（即第18块）复制到第0010行（即第2行），并置有效位为1，置标记为0000 001（表示信息取自主存第1块群）。

例4.5 假定主存和cache之间采用直接映射方式，块大小为一个字节。cache容量（指数据区）为4个字节，主存地址为32位，按字节编址。问：主存地址如何划分？根据程序访问的局部性原理说明块大小设置为1个字节时的缺陷。

解：块大小为1个字节，故块内无须寻址，块内地址位数为0。cache的数据区存放4个字节，共有4个行。因此，主存地址位数$n=32$，被划分为两个字段：标记位数$t=30$，行号位数$c=2$。

块大小设置为一个字节会产生两个方面的问题。

（1）邻近单元很可能被访问，但由于没有跟着该字节调入cache，因此邻近单元的访问会发生缺失。也就是说，块大小为一个字节时，程序访问的空间局部性没有被利用。

（2）在Cache行数不变的情况下，块太小使得映射到同一个cache行的主存块数增加，发生冲突的概率增大，引起频繁信息交换。

例4.6 假定主存和cache之间采用直接映射方式，块大小为16B。cache的数据区容量为64KB，主存地址为32位，按字节编址。问：主存地址如何划分？说明访存过程，并计

算 cache 总容量为多少?

解:cache 数据区容量为 $64\text{KB}=2^{16}\text{B}=2^{12}$ 行 $\times 2^{4}\text{B}$/行。

因为主存的每 2^{12} 块和 cache 的 2^{12} 行一一对应,所以可将主存的每 2^{12} 块看成一个块群,因而,得到主存空间划分为 $2^{32}\text{B}=2^{28}$ 块 $\times 2^{4}\text{B}$/块 $=2^{16}$ 块群 $\times$ 2^{12} 块/块群 $\times 2^{4}\text{B}$/块。

因此,主存地址位数 $n=32$,其中标记位数 $t=16$,行号位数 $c=12$,块内地址位数为 4。

主存地址的划分以及访存过程实现如图 4.25 所示。图中 Tag 表示标记字段,Index 表示 cache 行索引即行号,块内地址分为两部分:高 2 位 Word 为字偏移量、低 2 位 Byte 为字节偏移量。“Hit”表示命中。

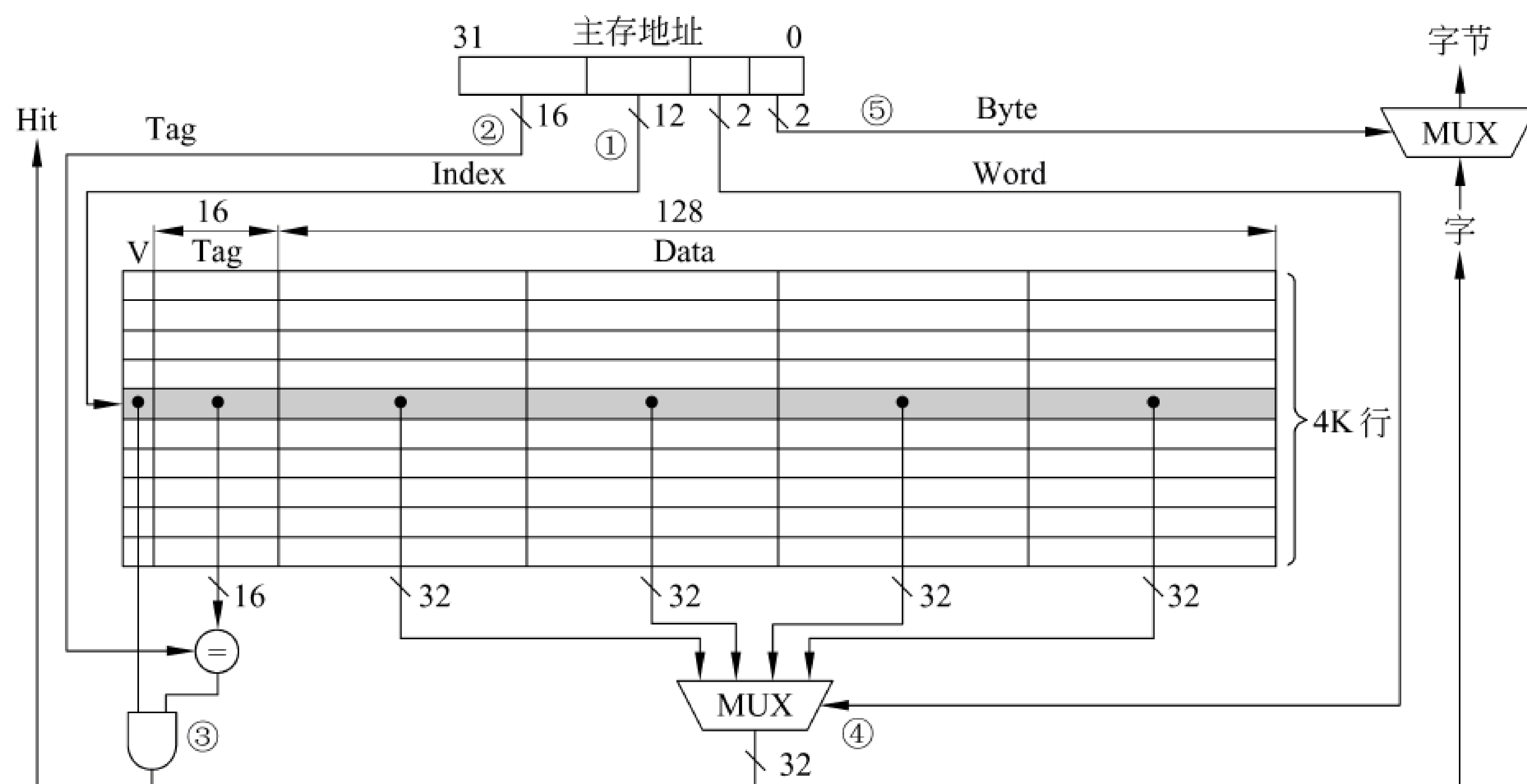


图 4.25　直接映射方式的实现

整个访存过程由硬件实现,分为以下 5 个步骤。

①根据 12 位 cache 行索引找到对应行;②将 16 位标记与对应行中的标记比较;③比较相等并有效位为 1 时,输出“Hit”为 1;④由两位字偏移量从 4 个 32 位字中选择一个字输出;⑤由两位字节偏移量从一个 32 位字中选择一个字节输出。CPU 在“Hit”为 1 的情况下,根据要访问的是字还是字节选择从第④步还是第⑤步得到结果。若“Hit”不为 1,则 CPU 要启动一次“cache 行读”总线事务操作,通过总线到主存读一块连续的信息到 cache 行中。

从图 4.25 中可看出每个 cache 行由一位有效位 V、16 位标记(Tag)和 4 个 32 位的数据(Data)组成,共有 $2^{12}=4\text{K}$ 行,因此,cache 的总容量为 $2^{12}\times(4\times 32+16+1)=4\text{K}\times 145=580\text{Kbits}=72.5\text{KB}$。其中,数据占总容量的 $64\text{KB}/72.5\text{KB}=88.3\%$。

直接映射的优点是容易实现,命中时间短,但由于多个块号“同余”的内存块只能映射到同一个 cache 行,当访问集中在“同余”内存块时,就会引起频繁的调进调出,即使其他 cache 行都空闲,也毫无帮助。很显然直接映射方式不够灵活,使得 cache 存储空间得不到充分利用,命中率较低。例如,上述例 4.4 中,若需将主存第 0 块与第 16 块同时调入 cache,由于它们都只对应 cache 第 0 行,即使其他行空闲,也总有一个主存块不能调入 cache,因此会产生频繁的调进调出。

2. 全相联映射

全相联映射的基本思想是一个主存块可装入 cache 任意一行中。全相联映射 cache 中,

每行的标记用于指出该行取自主存的哪个块。因为一个主存块可能在任意一行中，所以，需要比较所有 cache 行的标记，因此，主存地址中无须 cache 行索引，只有标记和块内地址两个字段。全相联映射方式下，只要有空闲 cache 行，就不会发生冲突，因而块冲突概率低。

例 4.7 假定主存和 cache 之间采用全相联映射，块大小为 512 字。cache 容量(指数据区)为 8K 字，主存地址空间为 1M 字。问：主存地址如何划分？用图表示主存块和 cache 行之间的映射关系，并说明 CPU 对主存单元 0240CH 的访问过程。

解：cache 数据区容量为 8K 字$=2^{13}$字$=2^4$ 行×512 字/行=16 行×512 字/行。

主存地址空间为 1M 字$=2^{20}$字$=2^{11}$块×512 字/块。

20 位的主存地址划分为两个字段：标记位数 $t=11$，块内地址位数为 9。

主存地址划分以及主存块和 cache 行的对应关系如图 4.26 所示。

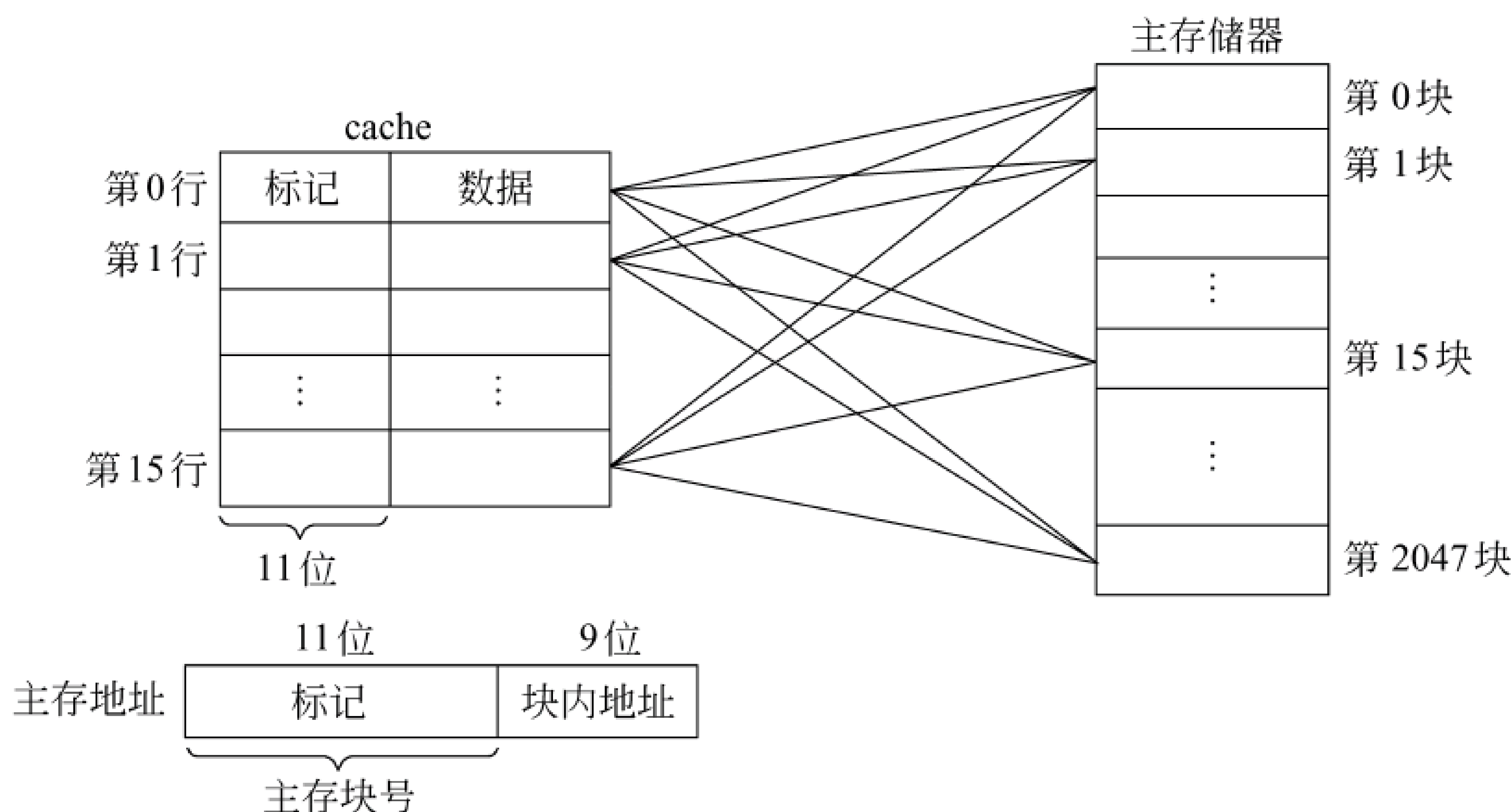


图 4.26 全相联映射方式下主存块和 cache 行对应关系

主存地址 0240CH 展开为二进制数为 0000 0010 0100 0000 1100，所以主存地址划分为：

0000 0010 010	0 0000 1100

访问 0240CH 单元的过程为：首先将高 11 位标记 0000 0010 010 与 cache 中每个行的标记进行比较，若有一个相等并且对应有效位为 1，则命中，此时 CPU 根据块内地址 0 0000 1100 从该行中取出信息；若都不相等，则不命中，此时，需要将 0240CH 单元所在的主存第 0000 0010 010 块(即第 18 块)复制到 cache 的任何一个空闲行中，并置有效位为 1，置标记为 0000 0010 010(表示信息取自主存第 18 块)。

为了加快比较的速度，通常每个 cache 行都设置一个比较器，比较器位数等于标记字段的位数。全相联 cache 访存时根据标记内容来访问 cache 行中的主存块，因而它查找主存块的过程是一种“按内容访问”的存取方式，因此，它是一种“相联存储器”。相联映射方式的时间开销和所用元件开销都较大，实现起来比较困难，不适合容量较大的 cache。

3. 组相联映射

前面介绍了全相联映射和直接映射，它们的优缺点正好相反，二者结合可以取长补短。因此将两种方式结合起来产生了组相联映射方式。

在直接映射方式中，那些块号“同余”的内存块只能被映射到同一个 cache 行中，从而引

起频繁的调进调出和较高的访问缺失率。一个直观的想法是,对于那些"同余"内存块,如果将映射的 cache 行从一个扩展为 2^s 个,那么,当访问第二个"同余"内存块时,因为有 2^s 个 cache 行可以使用,就不需要替换第一个在 cache 中的同余块;仅当 2^s 个 cache 行全部用完时,才需要考虑替换。根据这一思路,首先需要调整直接映射中 cache 行的结构设置,将其从原来的一维结构调整为 2^q 组×2^s 行/组的二维结构。将 cache 所有行分成 2^q 个大小相等的组,把主存块映射到 cache 固定组的 2^s 行中的任意一行。也即采用组间模映射、组内全映射的方式,映射关系如下。

cache 组号＝主存块号 mod cache 组数

例如,假定 8K 字的 cache 划分为:8 组×2 行/组×512 字/行,则主存第 100 块应映射到 cache 第 4 组的任意一行中,因为 100 mod 8＝4。

如此设置的 2^q 组×2^s 行/组的 cache 映射方式称为 2^s 路组相联映射,即 $s=1$ 为 2-路组相联;$s=2$ 为 4-路组相联;以此类推。通过对主存块号取模,使得每 2^q 个主存块与 2^q 个 cache 组一一对应,主存空间实际上被分成了若干组群,每个组群中有 2^q 个主存块。假设主存地址有 n 位,块内地址占 k 位,有 2^m 个组群,则 $n=m+q+k$,主存地址被划分为三个字段。

标记	cache 组号	块内地址

其中,高 m 位为标记,中间 q 位为组号(也称组索引),剩下的 k 位低位地址部分为块内地址。标记字段的含义表示当前地址所在的主存块位于主存哪个组群。

例如,假定 cache 数据区容量为 8KB,每个主存块大小为 32 字节,采用 2 路组相联,即每组有 2 行,则 cache 有 8KB/(32B×2)＝128 组,即 $q=7$,$s=1$。假定主存地址为 32 位,则 $m=32-7-5=20$,即主存共有 2^{20} 个组群,每个组群为 128 块,每块 32B,因而主存地址划分为:标记 20 位,组号 7 位,块内地址 5 位。

s 的选取决定了块冲突的概率和相联比较的复杂性。s 越大,则 cache 发生块冲突的概率越低,相联比较电路越复杂。选取适当的 s,可使组相联映射的成本比全相联低得多,而性能上仍可接近全相联方式。早几年,由于 cache 容量不大,所以通常 $s=1$ 或 2,即 2-路或 4-路组相联较常用,但随着技术的发展,cache 容量不断增加,s 的值有增大的趋势,目前有些处理器的 cache 也有采用 8-路或 16-路组相联的情况。

对于组相联 cache,CPU 访存过程为:首先根据访存地址中间的 q 位 cache 组号,直接找到对应的 cache 组;将对应 cache 组中每个行的标记与主存地址的高 m 位标记进行比较,若有一个相等并有效位为 1,则访问 cache"命中";根据主存地址中的块内地址,在对应 cache 行中存取信息;若都不相等或虽相等但有效位为 0,则"不命中",CPU 从主存中读出该地址所在的一块信息,送到 cache 对应组的任意一个空闲行中,将有效位置 1,并设置标记,同时将该地址中的内容送 CPU。实现组相联映射的硬件线路如图 4.27 所示。

图 4.27 所示的是采用 2-路组相联映射的 cache,整个访存过程如下。

① 根据主存地址中的 cache 组号找到对应组;②将地址中的标记与对应组中每个行的标记 Tag 进行比较;③将比较结果和有效位 V 相"与";④只要其中有一路比较相等并有效位为 1,就选中这一路 cache 行中的主存块,同时输出"Hit"为 1;⑤在"Hit"为1 的情况下,根据主存地址中的块内地址从选中的一块内取出对应单元的信息,若"Hit"不为 1,则 CPU 要到主存去读一块信息到 cache 行中。

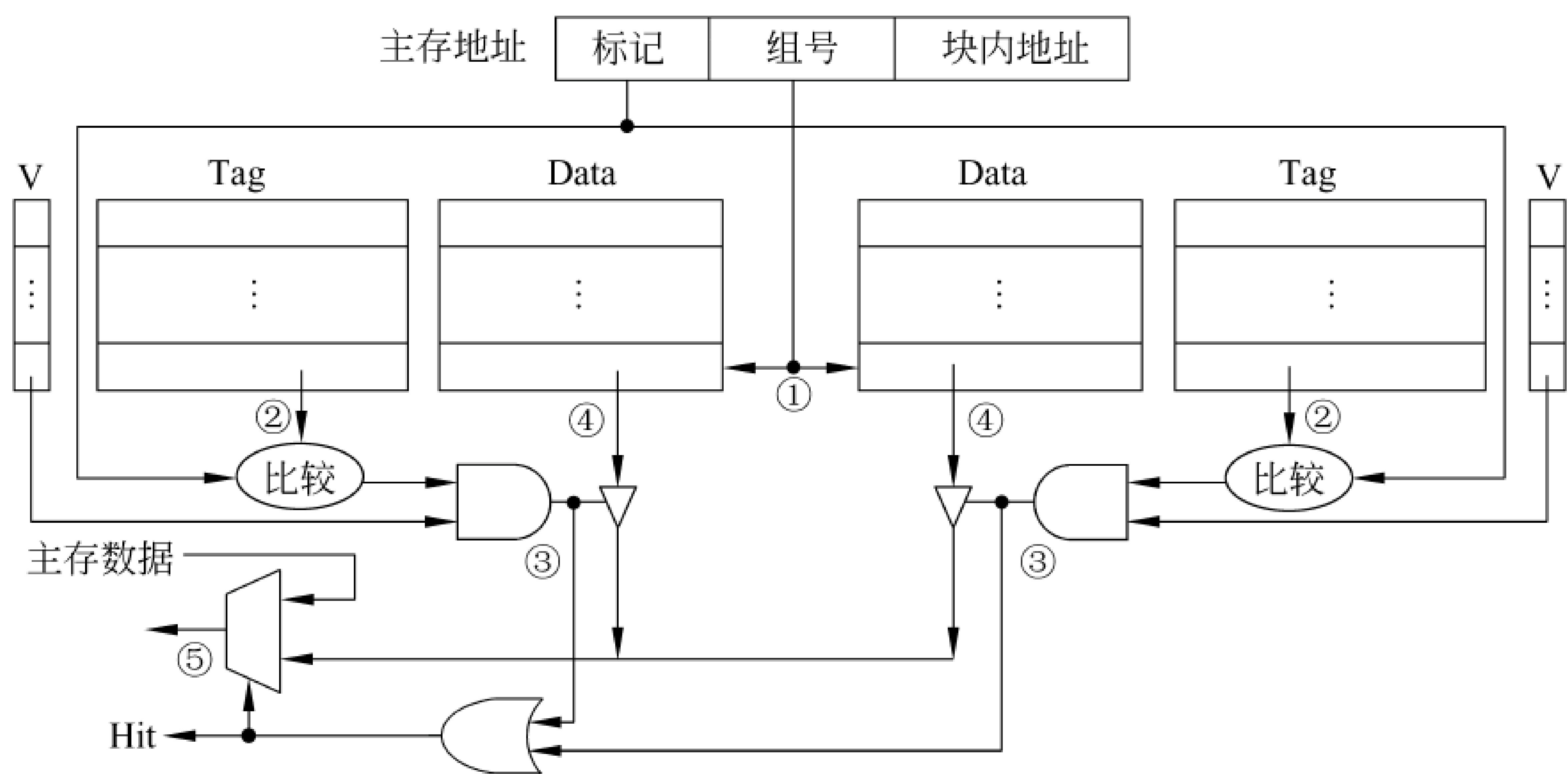


图 4.27　组相联映射方式的硬件实现

例 4.8　假定主存和 cache 之间采用 2 路组相联映射，块大小为 512 字。cache 容量（指数据区）为 8K 字，主存地址空间为 1M 字。问：主存地址如何划分？用图表示主存块和 cache 行之间的映射关系，并说明 CPU 对主存单元 0240CH 的访问过程。

解：cache 数据区容量为 8K 字$=2^{13}$字$=2^3$ 组$\times 2^1$ 行/组$\times 512$ 字/行。

主存地址空间为 1M 字$=2^{20}$字$=2^{11}$块$\times 512$ 字/块$=2^8$ 组群$\times 2^3$ 块/组群$\times 512$ 字/块。

所以，主存地址位数 $n=20$，标记位数 $m=8$，组号位数 $q=3$，块内地址位数为 9。

主存地址划分以及主存块和 cache 行的对应关系如图 4.28 所示。

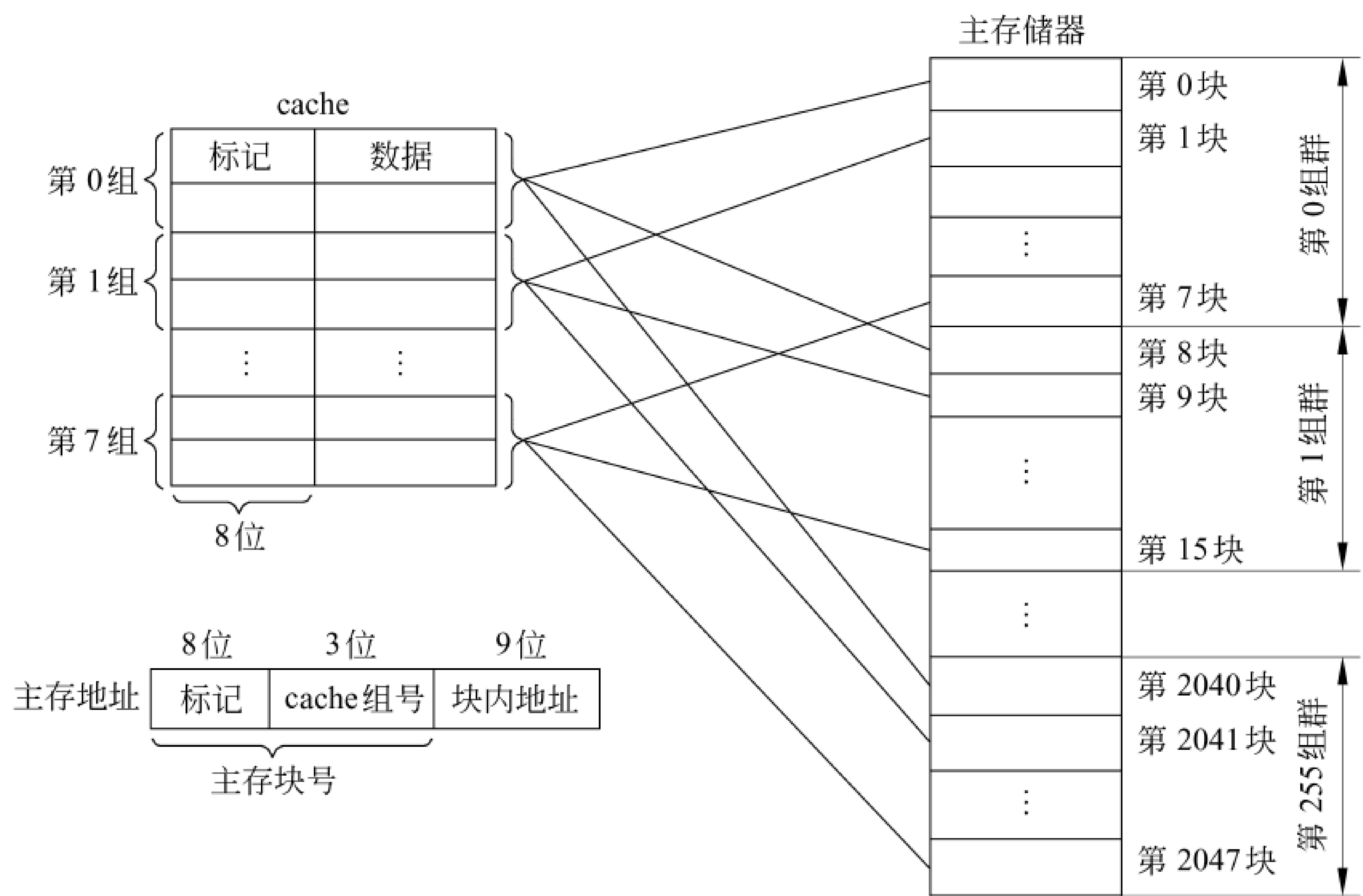


图 4.28　组相联映射方式下主存块和 cache 行对应关系

主存地址 0240CH 展开为二进制数为 0000 0010 0100 0000 1100，所以主存地址划分为：

0000 0010	010	0 0000 1100

访问 0240CH 单元的过程为：首先根据地址中间 3 位 010，找到 cache 第 2 组，将标记 0000 0010 与第 2 组中两个 cache 行的标记同时进行比较，若有一个相等并且有效位是 1，则命中。此时根据低 9 位块内地址从对应行中取出单元内容送 CPU；若都不相等或有一个相等但有效位为 0，则不命中。此时，将 0240CH 单元所在的主存第 0000 001 0010 块（即第 18 块）复制到第 010 组（即第 2 组）的任意一个空闲行中，并置有效位为 1，置标记为 0000 0010（表示数据来自第 2 组群）。

组相联映射方式结合了直接映射和全相联映射的优点。当 cache 的组数为 1 时，变为全相联映射；当每组只有一个 cache 行时，则变为直接映射。组相联映射的冲突概率比直接映射低，由于只有组内各行采用全相联映射，所以比较器的位数和个数都比全相联映射少，易于实现，查找速度也快得多。

4. 三种映射方式比较

对于一个主存块来说，三种映射方式下对应 cache 行的个数不同。直接映射是唯一映射，每个主存块只有一个固定行与之对应；全相联映射是任意映射，每个行都可对应；*N*-路组相联映射有 *N* 行对应。这种特性可用“关联度”来度量。也即关联度指一个主存块映射到 cache 中时可能存放的位置个数。因此，直接映射的关联度最低，为 1；全相联映射的关联度最高，为 cache 总行数；*N*-路组相联映射的关联度居中，为 *N*。

当 cache 大小、主存块大小一定时，关联度和命中率、命中时间、标记所占额外开销等有如下关系：

（1）关联度越低，命中率越低。因此直接映射命中率最低，全相联映射命中率最高。

（2）关联度越低，判断是否命中的开销越小，命中时间越短。因此，直接映射的命中时间最短，全相联映射的命中时间最长。

（3）关联度越低，标记所占额外空间开销越少。因此，直接映射额外空间开销最少，全相联映射额外空间开销最大。

例 4.9 假定主存地址为 32 位，按字节编址，主存块大小为 16 字节，cache 最多能存放 4K 个主存块数据，则在关联度分别为 1、2、4 和全相联方式下标记所占总位数是多少？

解：关联度为 1（直接映射）时，每组 1 行，共 4K 组，标记占 32－4－12＝16 位，总位数占 4K×16＝64K 位。

关联度为 2（2-路组相联）时，每组 2 行，共 2K 组，标记占 32－4－11＝17 位，总位数占 4K×17＝68K位。

关联度为 4（4-路组相联）时，每组 4 行，共 1K 组，标记占 32－4－10＝18 位，总位数占 4K×18＝72K位。

全相联时，整个为一组，每组 4K 行，标记占 32－4＝28 位，总位数占 4K×28＝112K 位。

4.6.4 cache 中主存块的替换算法

cache 行数比主存块数少得多，因此，往往多个主存块映射到同一个 cache 行中。当新的一个主存块复制到 cache 时，cache 中的对应行可能已经全部被占满，此时，必须选择淘汰掉一个 cache 行中的主存块。例如，对于例 4.8 中的 2-路组相联映射 cache，假定第 0 组的两个行分别被主存第 0 块和第 8 块占满，此时若需调入主存第 16 块，根据映射关系，它只能存放到 cache 第0 组，因此，已经在第 0 组的主存第 0 块和第 8 块中必须调出一块，到底调出

哪一块呢？这就是淘汰策略问题，也称为替换算法或替换策略。

常用的替换算法有：先进先出 FIFO（first-in-first-out）、最近最少用 LRU（least-recently used）、最不经常用 LFU（least-frequently used）和随机替换算法等。可以根据实现的难易程度以及是否能获得较高的命中率两方面来决定采用哪种算法。

1. 先进先出算法（FIFO）

FIFO 算法的基本思想是：总是选择最早装入 cache 的主存块被替换掉。这种算法实现起来较方便，但不能正确反映程序的访问局部性，因为最先进入的主存块也可能是目前经常要用的，因此，这种算法有可能产生较大的缺失率。

2. 最近最少用算法（LRU）

LRU 算法的基本思想是：总是选择近期最少使用的主存块被替换掉。这种算法能比较正确地反映程序的访问局部性，因为当前最少使用的块一般来说也是将来最少被访问的。但是，它的实现比 FIFO 算法要复杂一些。

下面用一个例子来说明 FIFO 算法和 LRU 算法的具体实现。为简化说明，以下假设组相联方式下不一定满足组大小是 2 的幂次，虽然这样假设与实际不符，但并不影响对实现原理的解释说明。

假定主存中的 5 块{1,2,3,4,5}映射到 cache 的同一组，对于主存块访问地址流{1,2,3,4,1,2,5,1,2,3,4,5}，在 3-路组相联和 4-路组相联方式下，采用 FIFO 算法的替换过程如图 4.29 所示。图中带 * 号的是最早复制到 cache 的主存块。同样的访问地址流，在 3-路组相联、4-路组相联和 5-路组相联的情况下，采用 LRU 算法的替换过程如图 4.30 所示。

	1	2	3	4	1	2	5	1	2	3	4	5
3行/组	1*	1*	1*	4	4	4*	5	5	5	5	5*	5*
		2	2	2*	1	1	1*	1*	1*	3	3	3
			3	3	3*	2	2	2	2	2*	4	4
								√	√			√
4行/组	1*	1*	1*	1*	1*	1*	5	5	5	5*	4	4
		2	2	2	2	2	2*	1	1	1	1*	5
			3	3	3	3	3	3*	2	2	2	2*
				4	4	4	4	4	4*	3	3	3
					√	√						

图 4.29　FIFO 替换算法示例

	1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	1	2	5	1	2	3	4
			1	2	3	4	1	2	5	1	2	3
				1	2	3	4	4	4	5	1	2
							3	3	3	4	5	1
3行/组								√	√			
4行/组					√	√		√	√			
5行/组					√	√		√	√	√	√	√

图 4.30　LRU 替换算法示例

从图 4.29 和图 4.30 可以看出,FIFO 算法的命中率并不随组的增大而提高;LRU 算法的平均命中率比 FIFO 算法高,而且算法命中率随组的增大而提高。这是因为在 FIFO 算法中,同一时刻小组内的主存块集合不一定是大组内主存块集合之子集,而在 LRU 算法中,同一时刻小组的块集合必然是大组块集合的子集。因此,在小组中命中时在大组中肯定命中,把满足这种特性的算法称为堆栈算法。因此,LRU 算法是堆栈算法,而 FIFO 算法不是堆栈算法。当然,并不是组越大越好,因为组内采用全相联映射,随着组的增大,实现的复杂性也会增加。

当程序中的分块局部化范围(即程序中某段时间集中访问的存储区)超过了 cache 组大小时,命中率可能变得很低。例如,假设上述例子中的访存地址流是 1,2,3,4,1,2,3,4,1,2,3,4,…,而 cache 每组只有 3 行,那么,不管是 FIFO 还是 LRU 算法,其命中率都为 0。这种现象称为颠簸(PingPong)或抖动(Thrashing)。

LRU 算法并不像图 4.30 所示的通过移动块来实现,可以通过给每个 cache 行设定一个计数器,用计数值来记录主存块的使用情况,通过硬件修改计数值,并根据计数值选择某个 cache 行中的主存块被淘汰,只要将被淘汰行的有效位清 0 即可。这个计数值称为 LRU 位,其位数与 cache 组的大小有关。2-路时有一位,4-路时有两位。

图 4.31 是上述例子中 4-路组相联的示例。图中左边的数字是对应 cache 行的计数值,右边的数字是存放在该行中的主存块号。

1		2		3		4		1		2		5		1		2		3		4		5	
0	1	1	1	2	1	3	1	0	1	1	1	2	1	0	1	1	1	2	1	3	1	0	5
		0	2	1	2	2	2	3	2	0	2	1	2	2	2	0	2	1	2	2	2	3	2
				0	3	1	3	2	3	3	3	0	5	1	5	2	5	3	5	0	4	1	4
						0	4	1	4	2	4	3	4	3	4	3	4	0	3	1	3	2	3

图 4.31 用计数器实现 LRU 算法

计数器的变化规则如下:

(1) 命中时,被访问的行的计数器清 0,比其低的计数器加 1,其余不变。

(2) 未命中且该组还有空闲行时,则新装入的行的计数器设为 0,其余全加 1。

(3) 未命中且该组无空闲行时,计数值为 3 的那一行中的主存块被淘汰,新装入的行的计数器设为 0,其余加 1。从计数器变化规则可以看出,计数值越高的行中的主存块越是最近最少用。

为简化上述 LRU 位计数的硬件实现,通常采用一种近似的 LRU 位计数方式来实现 LRU 算法。近似 LRU 计数方法并不严格按照上述计数器变化规则进行,而是大致区分哪些是新调入的主存块,哪些是较长时间未用的主存块,在较长时间未用的块中选择一个被替换出去。

3. 最不经常用算法(LFU)

LFU 算法的基本思想是:替换掉 cache 中引用次数最少的块。LFU 也用与每个行相关的计数器来实现。这种算法与 LRU 有点类似,但不完全相同。

4. 随机替换算法(Random)

从候选行的主存块中随机选取一个淘汰掉,与使用情况无关。模拟试验表明,随机替换

算法在性能上只稍逊于基于使用情况的算法，而且代价低。

例 4.10 假定主存空间大小为 32K×16 位，按字编址，每字 16 位。cache 采用 4-路组相联映射方式，数据区大小为 4K 字，主存块大小为 64 字。假定 cache 开始为空，处理器按顺序访问主存单元 0，1，…，4351，一共重复访问 10 次。假设 cache 比主存快 10 倍，采用 LRU 替换算法。试分析采用 cache 后速度提高了多少？

解：主存空间大小为 32K 字＝512 块×64 字/块。

cache 数据区容量为 4K 字＝16 组×4 行/组×64 字/行。

所以，cache 共有 64 行，分成 16 组，每组 4 行。

因为每块为 64 字，4352/64＝68，所以主存单元 0～4351 应该对应前 68 块（第 0～67 块），即处理器的访问过程是对主存前 68 块连续访问 10 次。

图 4.32 给出了前两次循环的主存块替换情况，图中列方向是 cache 的 16 个组，行方向是每组的 4 个 cache 行。根据组相联映射的特点，cache 行和主存块之间的映射关系如下：主存第 0～15 块分别对应 cache 第 0～15 组，可以放在对应组的任一行中，在此，假定按顺序存放在第 0 行；主存第 16～31 块也分别对应 cache 的第 0～15 组，存放在第 1 行；同理，主存第 32～47 块分别放到 cache 第 0～15 组的第 2 行；第 48～63 块分别放到 cache 第 0～15 组的第 3 行。这样，第 0～63 块都没有冲突，每块都是第一个字在 cache 中没有找到，调到 cache 对应组的某一行后，其余每个字都能在 cache 中找到。因此每一块只有第一字未命中，其余字都命中。

	第 0 行	第 1 行	第 2 行	第 3 行
第 0 组	0 / 64 / 48	16 / 0 / 64	32 / 16	48 / 32
第 1 组	1 / 65 / 49	17 / 1 / 65	33 / 17	49 / 33
第 2 组	2 / 66 / 50	18 / 2 / 66	34 / 18	50 / 34
第 3 组	3 / 67 / 51	19 / 3 / 97	35 / 19	51 / 35
第 4 组	4	20	36	52
⋮	⋮	⋮	⋮	⋮
第 15 组	15	31	47	63

图 4.32　例 4.9 中主存块的替换情况

主存的第 64～67 块分别对应 cache 的第 0～3 组，此时，这 4 组的 4 行都不空闲，所以要选择一个字块被淘汰出来。因为采用 LRU 算法，所以，分别将最近最少用的第 0～3 块从第 0～3 组的第 0 行中替换出来。再把第 64～67 块分别放到对应 cache 行中，每块也都是第一个字在 cache 中没有找到，调入后其余每字都能在 cache 中找到。

综上所述，第一次循环时，对于所有 68 块都只有第一字未命中，其余字都命中。

以后 9 次循环中，因为 cache 第 4～15 组中的 4×12＝48 个 cache 行内的主存块一直没有被替换，所以只有 68－48＝20 个主存块的第一字未命中，其余都命中。

访问总次数为 4352×10＝43520，其中，未命中次数为 68＋9×20＝248。

命中率 p＝(43520－248)/43520＝99.43%。

假定 cache 和主存的访问时间分别为 t_c 和 t_m，根据题意可知 $t_m=10t_c$。采用 cache 后，

cache-主存层次的平均访问时间为：$t_a = t_c + (1-p)t_m = t_c + (1-p) \times 10t_c$。

因此，采用 cache 后速度提高的倍数为

$$t_m / t_a = 10t_c / (t_c + (1-p) \times 10t_c) = 10/(1 + (1-p) \times 10) \approx 9.5。$$

4.6.5 cache 的一致性问题

因为 cache 中的内容是主存块副本，当对 cache 中的内容进行更新时，就存在 cache 和主存如何保持一致的问题。除此之外，以下情况也会出现“cache 一致性问题”。

① 当多个设备都允许访问主存时。例如，像磁盘这类高速 I/O 设备可通过 DMA 方式直接读写主存，如果 cache 中的内容被修改而主存块没有更新的话，则从主存传送到 I/O 设备的内容就无效；若 I/O 设备修改了主存块的内容，则对应 cache 行中的内容就无效。

② 当多个 CPU 都带有各自的 cache 而共享主存时。在多 CPU 系统中，若某个 CPU 修改了自身 cache 中的内容，则对应的主存块和其他 CPU 中对应的 cache 行的内容都变为无效。

解决 cache 一致性问题的关键是处理好写操作，通常有两种写操作方式。

1. 全写法(write through)

基本做法是：写操作时，若写命中，则同时写 cache 和主存；若写不命中，则有以下两种处理方式。

(1) 写分配法(write allocate)。先在主存块中更新相应存储单元，然后分配一个 cache 行，将更新后的主存块装入到分配的 cache 行中。这种方式可以充分利用空间局部性，但每次写不命中都要从主存读一个块到 cache 中，增加了读主存块的开销。

(2) 非写分配法(not write allocate)。仅更新主存单元而不装入主存块到 cache 中。这种方式可以减少读入主存块的时间，但没有很好利用空间局部性。

由此可见，该方式实际上采用的是对主存块信息及其所有副本信息全都直接同步更新的做法，因此，该方式通常也被称为通写法或直写法，也有教材称之为写直达法。

显然，全写法在替换时不必将被替换的 cache 内容写回主存，而且 cache 和主存的一致性能得到充分保证。但是，这种方法会大大增加写操作的开销。例如，假定一次写主存需要 100 个 CPU 时钟周期，那么 10%的存数指令就使得 CPI 增加了 100×10%=10 个时钟。

为了减少写主存的开销，通常在 cache 和主存之间加一个写缓冲(write buffer)。在 CPU 写 cache 的同时，也将信息写入写缓冲，然后由存储控制器将缓冲中的内容写主存。写缓冲是一个 FIFO 队列，一般有 4 项，在写操作频率不是很高的情况下，效果较好。但是，如果写操作频繁发生，则会使写缓冲饱和而发生阻塞。

2. 回写法(write back)

基本做法是：当 CPU 执行写操作时，若写命中，则信息只被写入 cache 而不被写入主存；若写不命中，则在 cache 中分配一行，将主存块调入该 cache 行中并更新相应单元的内容。因此，该方式下在写不命中时，通常采用写分配法进行写操作。

由此可见，该方式实际上采用的是回头再写或最后一次性写的做法，因此，该方式通常被称为回写法或一次性写方式，也有教材称之为写回法。

在 CPU 执行写操作时，回写法不会更新主存单元，只有当 cache 行中的主存块被替换时，才将该块内容一次性写回主存。这种方式的好处在于减少了写主存的次数，因而大大降低了主存带宽需求。为了减少写回主存块的开销，每个 cache 行设置了一个修改位(dirty

bit)，有时也称为“脏位”。若修改位为1，则说明对应cache行中的主存块被修改过，替换时需要写回主存；若修改位为0，则说明对应主存块未被修改过，替换时无须写回主存。

由于回写法没有同步更新cache和主存内容，所以存在cache和主存内容不一致而带来的潜在隐患。通常需要其他的同步机制来保证存储信息的一致性。

4.6.6 cache性能评估

计算机性能最直接的度量方式就是CPU时间。执行一个程序所花的CPU时间应该等于CPU执行时间和等待主存访问时间之和。当发生cache缺失时，需要等待主存访问，此时，CPU处于阻塞状态。因此，

CPU时间=(CPU执行时钟数+cache缺失引起阻塞的时钟数)×时钟周期

cache缺失引起阻塞的时钟数=读操作阻塞时钟数+写操作阻塞时钟数

读操作阻塞时钟数=程序中读操作次数×读缺失率×读缺失损失

对于写操作来说，情况较复杂。根据不同的写策略，其阻塞时钟数的计算方式不同。

(1) 回写方式下，替换时需要一次性写回一个块，故会产生一些附加写回阻塞。

写操作阻塞时钟数=程序中写操作次数×写缺失率×写缺失损失+写回阻塞

(2) 全写方式下，包括写缺失阻塞和写缓冲(write buffer)阻塞两部分。

写操作阻塞时钟数=程序中写操作次数×写缺失率×写缺失损失+写缓冲阻塞

假定写回阻塞和写缓冲阻塞忽略不计，则可将读操作和写操作综合考虑，得到如下公式。

cache缺失引起阻塞的时钟数 =程序的访存次数×缺失率×缺失损失

=程序的指令条数×(缺失数/指令)×缺失损失

例4.11 假设某计算机中只有一级cache，并将指令和数据分别存放在code cache和data cache中。其Code Cache和Data Cache的缺失率分别为1%和4%。假定在没有任何访存阻塞时的CPI为1，缺失损失为200个时钟周期。假定访存指令(load和store)的使用频度为36%，则使用缺失率为0的cache时，处理器速度会快多少?

解：设程序运行中执行指令I条，则访问指令缺失时所用时钟周期数为$I\times1\%\times200=2.0\times I$。已知访存指令的使用频度为36%，所以访问数据缺失时所用时钟周期数为$I\times36\%\times4\%\times200=2.88\times I$。

因为在一条指令执行过程中取指令和访问数据总是串行进行的，所以，两者的阻塞时钟数应该相加，即指令缺失和数据缺失时的总阻塞时钟数为$2.0\times I+2.88\times I=4.88\times I$，也即平均每条指令要有4.88个时钟周期处在访存阻塞状态，因此，由于访存阻塞而使得CPI数从没有访存阻塞时的1增大到1+ 4.88=5.88。故如果cache不发生缺失(即缺失率为0)，则处理器速度会快5.88/1=5.88倍。

访存阻塞所花时间占整个执行时间的比例为4.88/5.88≈83%。

对上述例子进一步分析，可以得到处理器的性能与cache性能之间的依赖关系。分以下两个方面来考虑。

(1) 假设上例中没有任何访存阻塞时的CPI为2，时钟宽度不变，则：

访存阻塞使得CPI数从2增加到2+4.88=6.88。

如果cache不发生缺失，则处理器速度会快6.88/2=3.44倍。

访存阻塞所用时间占整个执行时间的比例为 4.88/6.88≈71%，小于 83%。

因此，可得出结论，CPI 越小，cache 缺失引起的阻塞对系统总体性能的影响越大。

(2) 假定上例中时钟频率加倍，CPI 不变，则：

主存速度不太可能改变，故绝对时间不变，所以缺失损失变为 400 个时钟周期。

访问指令和数据缺失时引起的总阻塞时钟数为(1%×400)+36%×(4%×400)=9.76。

因此访存阻塞使得 CPI 数从 1 增大到 1+9.76=10.76。

由此可知，时钟频率快的机器的性能只是较慢机器的 5.88/(10.76/2)≈1.1 倍。

如果没有 cache 缺失的话，应该是 2 倍。

由此可得出结论，CPU 时钟频率越高，cache 缺失损失就越大。

上述两个方面的例子说明：处理器性能越高，cache 的性能就越重要！

*4.6.7 影响 cache 性能的因素

决定系统访存性能的重要因素之一是 cache 命中率，它与许多因素有关。前面曾讲过，命中率与关联度有关。除此之外，最明显的就是和 cache 容量有关。显然，cache 容量越大，命中率就越高。此外，命中率还与主存块的大小有一定关系。采用大的交换单位能很好地利用空间局部性，但是，较大的主存块需要用较多的时间来存取，因此，缺失损失会变大；而且，主存块越大，cache 的总行数就越少，因而缺失率上升。由此可见，主存块的大小必须适中，不能太大，也不能太小。

除了上述提到的这些因素外，设计 cache 时，还要考虑采用单级还是多级 cache、数据 cache 和指令 cache 是分开还是合在一起、主存—总线—cache—CPU 之间采用什么架构等，甚至主存 DRAM 芯片的内部结构、存储器总线的总线事务类型等，也都与 cache 设计有关，都会影响系统总体性能。下面对这些问题进行简单分析说明。

1. 单级/多级 cache、联合/分离 cache 的选择问题

cache 技术刚被引入时，通常采用单级 cache，只有一个 CPU 片内 cache。近年来，多级 cache 系统已成为主流，同时使用片内 L1 cache 和 L2 cache，甚至 L3 cache。L2 cache 和 L3 cache 可以是在 CPU 芯片内也可以在 CPU 芯片外。通常 L1 cache 采用分离 cache，即数据 cache 和指令 cache 分开设置。L2 cache 和 L3 cache 通常为联合 cache 方式，即数据和指令放在一个 cache 中。

在一个采用两级 cache 的系统中，若 CPU 访问 L1 cache 缺失，则先从 L2 cache 中找，若 L2 cache 包含所请求信息，则缺失损失为 L2 cache 的访问时间，这比访问主存要快得多；若 L2 cache 不包含所请求信息，则需从主存取信息并同时送 L1 cache 和 L2 cache，此时的缺失损失较大。

在多级 cache 中，有全局缺失率和局部缺失率两种不同的概念。全局缺失率是指在所有级 cache 中都缺失的访问次数占总访问次数的比率；局部缺失是指在某级 cache 中缺失的访问次数占对该级 cache 的总访问次数的比率。例如，对于两级 cache，若 CPU 总的访存次数为 100，在 L1 cache 命中的次数为 94，剩下的 6 次中在 L2 cache 命中的次数为 5，只有一次需要访问主存，则全局缺失率为 1%，L1 cache 和 L2 cache 的局部缺失率分别为 6%和 16.7%。

下面举例说明增加 L2 cache 后对系统性能的提升情况。

例 4.12 假定某处理器的时钟频率为 1.2GHz，当 L1 cache 无缺失时的 CPI 为 1。访

问一次主存的时间为100ns(包括所有缺失处理),L1 cache的局部缺失率为2%。若增加一个L2 cache,并假定L2 cache的访问时间为5ns,而且其容量足够大到使全局缺失率仅为0.5%,问:增加L2 cache后处理器执行程序的效率提高了多少?

解:若仅有L1 cache,则仅发生L1 cache缺失,其缺失损失为100ns×1.2GHz=120个时钟周期;此时,由于访存阻塞而使得CPI从1变为1+120×2%=3.4。

若同时又有L2 cache,则存在以下两种情况。

(1) 若L1 cache缺失而L2 cache命中,则缺失损失为5ns×1.2GHz=6个时钟周期。

(2) 若L1和L2 cache都缺失,则需访问主存,缺失损失为100ns×1.2GHz=120个时钟周期。

因此,由于访存阻塞而使得CPI数从1变为1+6×(2%−0.5%)+120×0.5%=1.69。

综上可知,增加L2 cache使处理器执行程序效率提高了3.4/1.69≈2倍。

由于多级cache中各级cache所处的位置不同,使得对它们的设计目标有所不同。例如,假定是两级cache,那么,对于L1 cache,通常更关注速度而不要求有很高的命中率,因为,即使不命中,还可以到L2 cache中访问,L2 cache的速度比主存速度快得多;而对于L2 cache,则要求尽量提高其命中率,因为若不命中,则必须到慢速的主存中访问,其缺失损失会很大。

2. 主存—总线—cache间的连接结构问题

在主存和cache之间传输的单位是主存块,要使缺失损失最小,必须在主存、总线和cache之间构建快速的传输通道。什么样的连接结构才能使主存块在主存和cache之间的传输速度最快呢?

为了计算主存块传送到cache所用的时间,必须先了解CPU从主存取一块信息到cache的过程。从主存读一块数据到cache,一般包含以下三个阶段:

(1) 发送地址和读命令到主存:假定用1个时钟周期;

(2) 主存准备好一个数据:假定用10个时钟周期;

(3) 从总线传送一个数据:假定用1个时钟周期。

主存、总线和cache之间可以有三种连接方式:①窄形结构,即在主存、总线和cache之间每次按一个字的宽度进行传送;②宽形结构,即在它们之间每次传送多个字;③交叉存储器结构,主存采用多模块交叉存取方式,总线和cache之间每次按一个字的宽度进行传送。假定一个主存块有4个字,那么对于这三种结构,其缺失损失各是多少呢?

图4.33给出了三种方式下的主存块传送过程。图4.33(a)对应于窄形结构,连续进行“送地址-读出-传送”4次,每次一个字,其缺失损失为4×(1+10+1)=48个时钟周期;图4.33(b)对应于宽度为两个字的宽形结构,连续进行“送地址-读出-传送”两次,每次两个字,其缺失损失为2×(1+10+1)=24个时钟周期;假定宽形结构的宽度为4个字,则只要进行“送地址-读出-传送”一次,其缺失损失为1×(1+10+1)=12个时钟周期;图4.33(c)对应采用4体交叉存储结构,在首地址送出后,每隔一个时钟启动一个存储模块,第1个模块用10个时钟周期准备好第1个字,然后在总线上传送第1个字,同时,第2个模块准备第2个字,总线上传输第2个字的同时,第3个模块准备第3个字,总线上传输第3个字的同时,第4个模块准备第4个字,最后总线传送第4个字。因此,其缺失损失为1+1×10+4×1=15个时钟周期。由此可见,交叉存储结构的性价比最好。

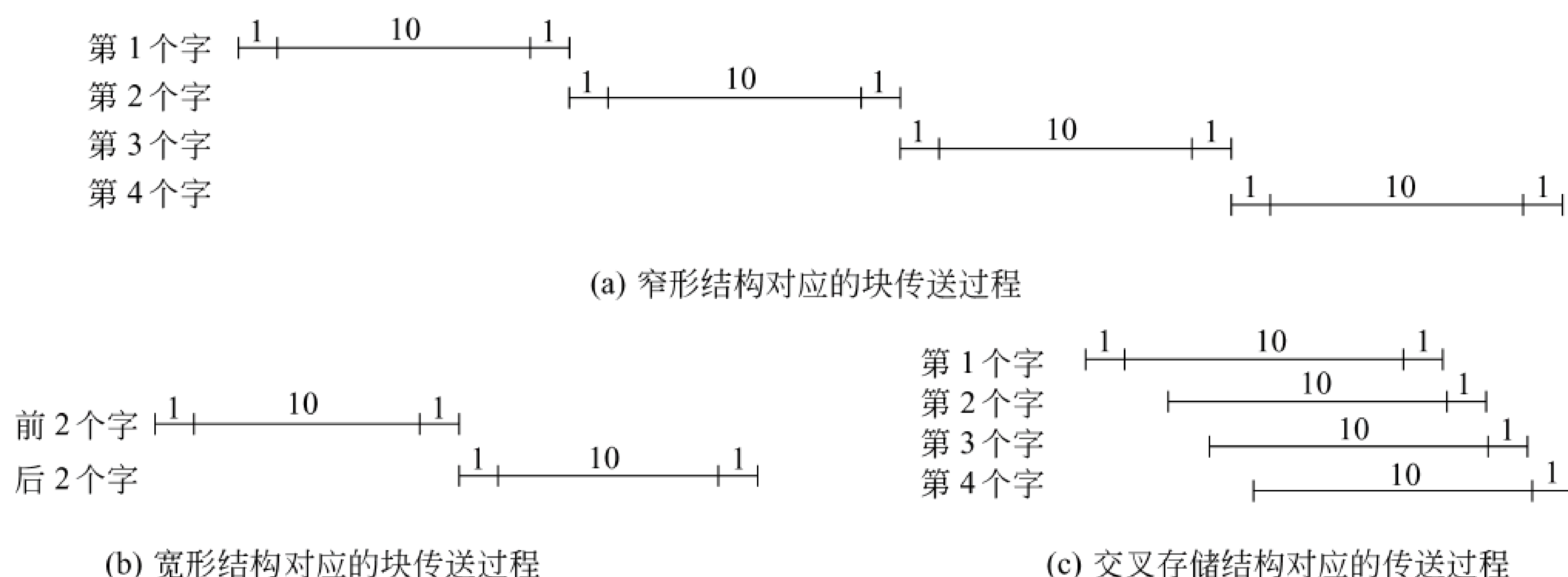


图 4.33　主存块在主存—总线—cache 之间的传送过程

3. DRAM 结构、总线事务类型与 cache 的配合问题

指令执行过程中，若发生 cache 缺失，则到主存取数据或指令，而主存是由 DRAM 芯片实现的，并且每次缺失时，要从 DRAM 中读取一块信息到 cache。因此，如何合理设计 DRAM 结构，如何使存储器总线在一次总线事务中高效地传输一个主存块等，都是需要和 cache 设计统一考虑的问题。

如图 4.34 所示是一台计算机中的内存条在存储器总线上的排列示意图，图 4.35 所示是一个内存条上 DRAM 芯片的排列示意图。

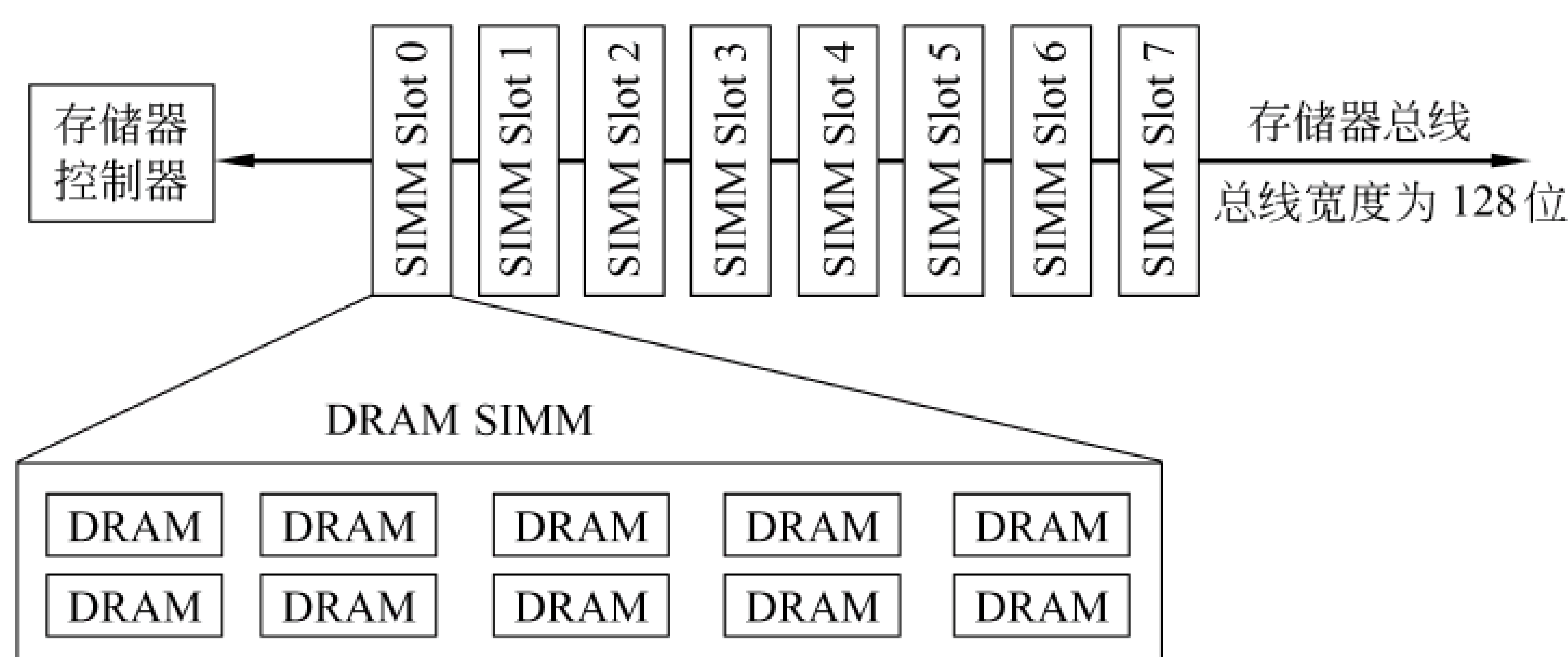


图 4.34　内存条排列示意图

图 4.34 所示的存储器总线宽度为 128 位，连接在其上的每个内存条一次最多能读出 128 位数据。每个内存条上排列有多个 DRAM 芯片，如图 4.35 所示。可用 16 个 2Mb 的 DRAM 芯片配置成一个 4MB 的内存条，每个芯片内有一个 512×8 的 SRAM 行缓冲，16 个芯片共 8KB 缓冲。每个芯片有 512 行×512 列，并有 8 个位平面，每次读写各芯片内同行同列的 8 位，共 16×8＝128 位。当 CPU 访问一块连续的主存区(即行地址相同)时，可直接从行缓冲读取，行缓冲用 SRAM 实现，速度极快。当 cache 缺失而要求从主存读一块信息到 cache 时，只要给定一个首地址，采用突发传输方式就可以在一次总线事务中完成一个主存块的传输。特别是当采用 DDR SDRAM、DDR2 SDRAM 或 DDR3 SDRAM 芯片时，在芯片内部采用交叉多数据预取，并在存储器总线上采用时钟上升沿和下降沿各传送一次的方式，使得从主存到 cache 的数据块传送效率更高。

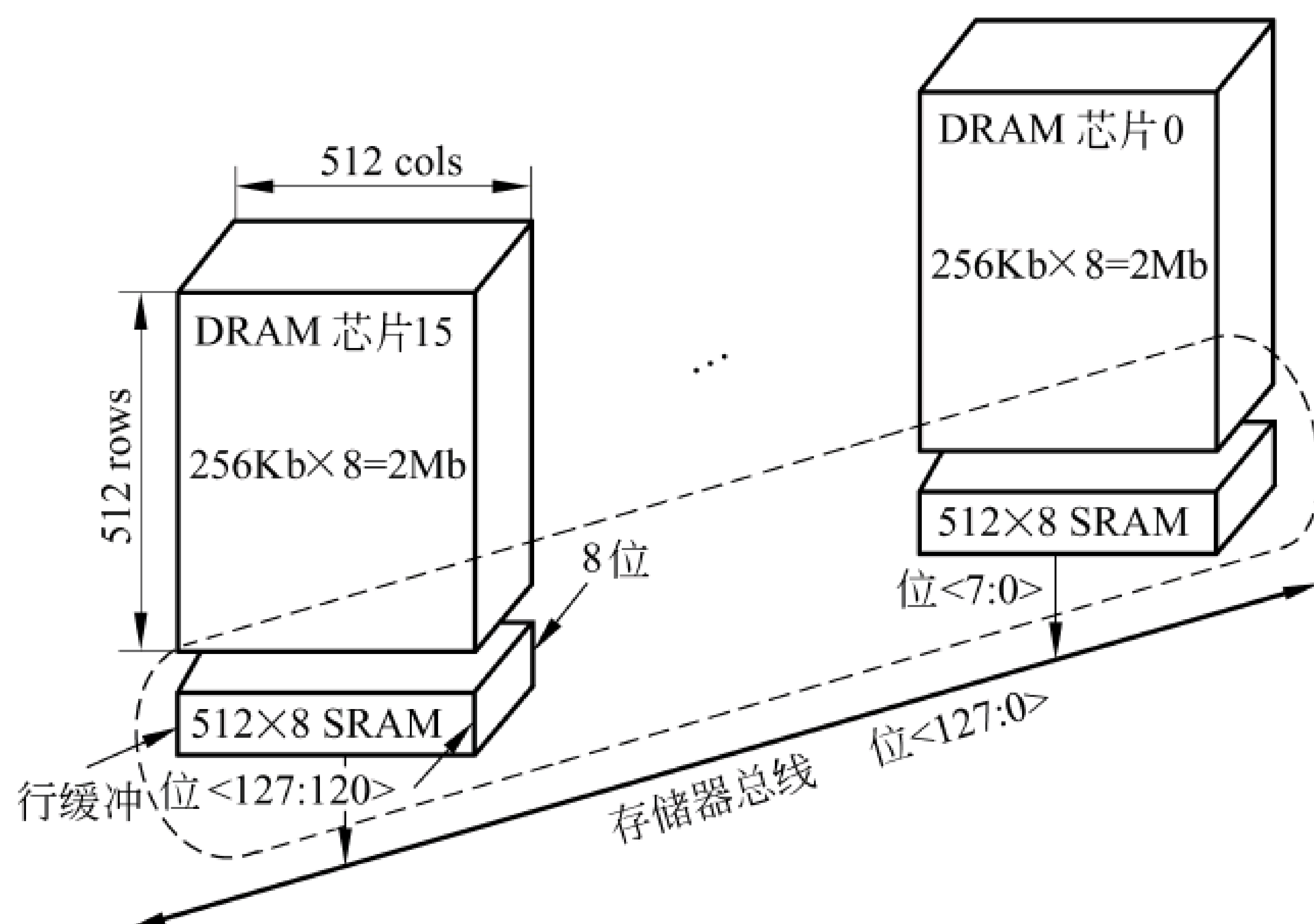


图 4.35　内存条中芯片排列示意图

* 4.6.8　cache 结构举例

现代计算机系统中几乎都使用 cache 机制，以下以 Pentium 和 Pentium 4 微处理器中的 cache 为例来说明具体的 cache 结构。

Pentium 微处理器在芯片内集成了一个代码 cache 和一个数据 cache。Pentium 的核心处理部件是两个整数 ALU 和一套浮点运算部件。两个整数 ALU 可并行工作，数据 cache 直接向它们提供操作数，而代码 cache 对 CPU 来说是只读的，它给指令预取缓冲器填充预取指令队列。数据 cache 采用双端口结构，每个端口 32 位，各通过一组 32 位总线分别与一个整数 ALU 相连。两个端口可以合并为一个 64 位端口，通过 64 位总线与浮点部件相连。

片内 cache 采用两路组相联结构，如图 4.36 所示，共 128 组，每组 2 行，每行 8 个双字[①]，共 4×8＝32 字节，因此每路有 4K 字节的容量，共 8K 字节。

片内 cache 的两路中各有一个目录表，每个表有 128 个记录项，每个记录项由 20 位的“标记”和 2 位的“状态”组成，共有 4 种不同状态，用于 cache 一致性协议（称为 MESI 协议[②]）。当一个主存块调入 cache 后，就将其 32 位地址中的高 20 位标记填入目录表中对应组（目录 0 或目录 1）的一个记录项中。

片内 cache 采用 LRU 替换策略，每组有一个 LRU 位，用来表示该组哪一路中的 cache 行被替换。

数据 cache 采用回写策略，不过可动态重构成全写方式。另外，Pentium 处理器还有两条单独的指令来清除或回写 cache。

---

① IA32 体系结构中字为 16 位，双字为 32 位，即 4 个字节。

② MESI 协议用于解决 cache 一致性问题，将每个 cache 行的状态分为更新（Modified）、独占（Exclusive）、共享（Shared）、无效（Invalid）4 种，通过对状态转换进行控制来实现数据的一致性。更新表示在 cache 行中的信息已被修改过；独占表示在其他 cache 中没有副本；共享表示在其他 cache 中有副本；无效表示 cache 行的信息无效，是空闲行，可存放新的主存块。

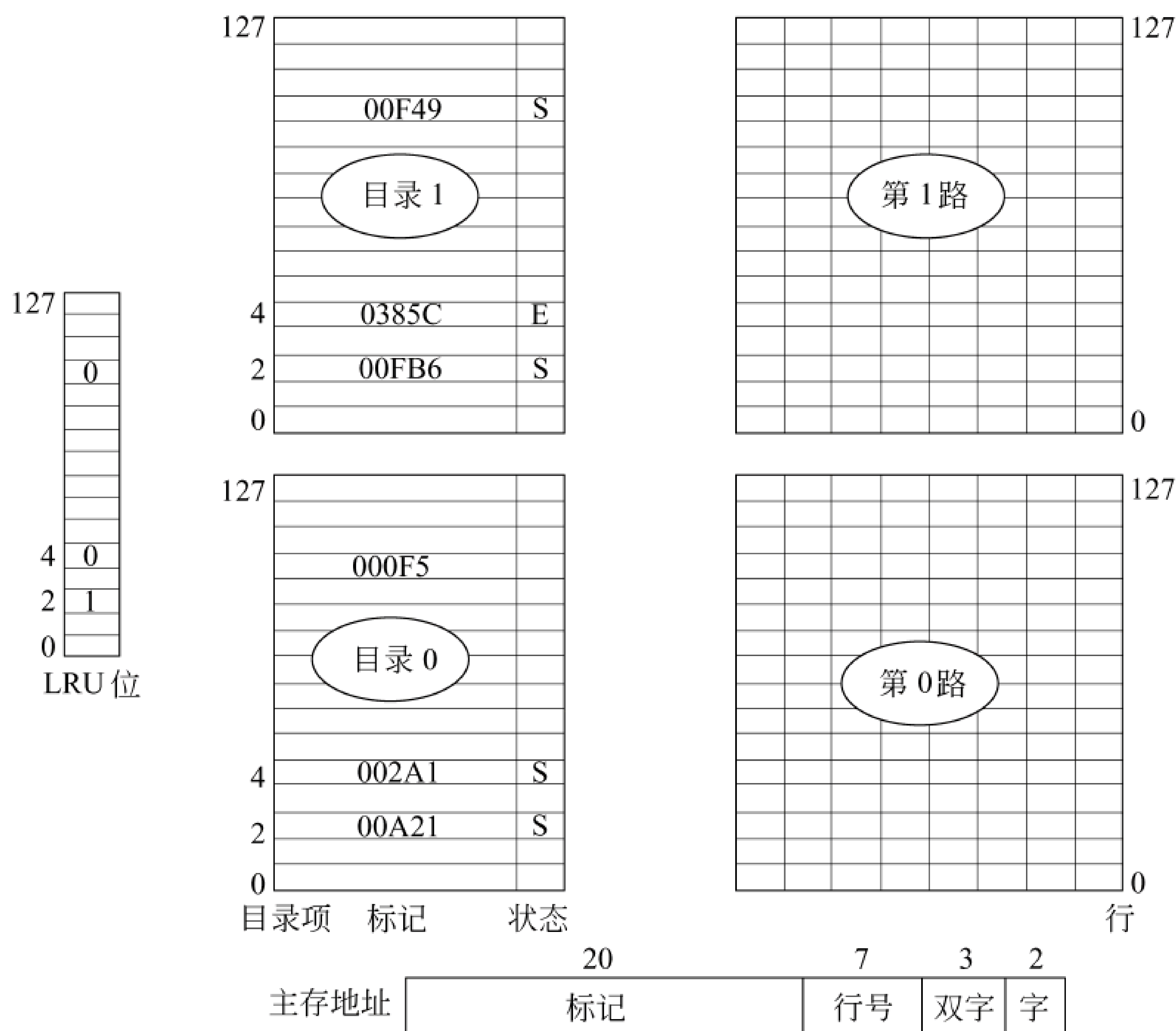


图 4.36　Pentium 处理器片内 cache 结构

Pentium 处理器支持采用片外的二级 cache。片外 L2 cache 可配置为 256KB 或 512KB，也采用两路组相联方式，每行 32、64 或 128 字节。

Pentium 4 处理器芯片内集成了一个 L2 cache 和两个 L1 cache。L2 cache 是联合 cache，数据和指令存放在一起，所有从主存获取的指令和数据都先送到 L2 cache 中。它有三个端口，一个对外，两个对内。对外的端口通过预取控制逻辑和总线接口部件和处理器总线相连，用来和主存交换信息；对内的端口中，一个以 256 位位宽与 L1 数据 cache 相连；另一个以 64 位位宽与指令预取部件相连，由指令预取部件取出指令，送指令译码器，指令译码器再将指令转换为微操作序列，送到指令 cache 中，Intel 称该指令 cache 为踪迹高速缓存(Trace Cache，TC)，其中存放的并不是指令，而是指令对应的微操作序列。有关细节可参考第 7.4.3 节。

4.7　虚拟存储器

目前计算机主存主要由 DRAM 芯片构成，由于技术和成本等原因，主存的存储容量受到限制，并且各种不同计算机所配置的物理内存容量多半也不相同，而程序设计时人们显然不希望受到特定计算机的物理内存大小的制约，因此，如何解决这两者之间的矛盾是需要解决的一个重要问题；此外，现代操作系统都支持多道程序运行，如何让多个程序有效而安全地共享主存是需要解决的另一个问题。

为了解决上述两个问题，在计算机中采用了虚拟存储技术：程序员在一个不受物理内

存空间限制并且比物理内存空间大得多的虚拟的逻辑地址空间中编写程序，就好像每个程序都独立拥有一个巨大的存储空间一样。程序执行过程中，把当前执行到的一部分程序和相应的数据调入主存，其他暂不用的部分暂时存放在磁盘上。这种借用外存为程序提供的很大的虚拟存储空间称为虚拟存储器。

指令执行时，通过硬件将指令中的逻辑地址（也称虚拟地址或虚地址）转化为主存的物理地址（也称主存地址或实地址），在地址转换过程中检查是否发生访问信息缺失、地址越界或访问越权，若发生信息缺失，则由操作系统进行主存和磁盘之间的信息交换。若发生地址越界或访问越权，则由操作系统进行存储访问的异常处理。由此可以看出，虚拟存储技术既解决了编程空间受限的问题，又解决了多道程序共享主存带来的安全等问题。

虚拟存储器机制由硬件与操作系统共同协作实现，涉及到计算机系统许多层面，包括操作系统中的许多概念，如进程、进程的上下文切换、存储器管理、虚拟地址空间、缺页处理等。因此，下面从操作系统的有关概念开始介绍。

* 4.7.1　进程与进程的上下文切换

用户程序通过操作系统中进程、虚拟存储器和文件等机制使用硬件资源，使得每个用户程序在运行时产生错觉，以为所有系统资源都被自己独占使用，且处理器始终执行本程序的一条条指令。

进程是操作系统对处理器中运行的程序的一种抽象。现代的多任务操作系统中，通常可同时运行很多进程，但每个进程都好像自己独占使用计算机资源。实际上，操作系统通过处理器调度让处理器交替执行多个进程中的指令，实现不同进程中指令交替执行的机制称为“上下文切换(context switching)”。

进程的物理实体和支持进程运行的环境合称为进程的上下文。由用户的程序块、数据块和堆栈等组成的用户区地址空间，被称为用户级上下文；由进程标识信息、现场信息、控制信息和系统内核栈等组成的系统区地址空间，被称为系统级上下文；此外，还包括处理器中各个寄存器的内容，被称为寄存器上下文。在进行进程上下文切换时，操作系统把换下进程的寄存器上下文保存到系统级上下文的现场信息位置。用户级上下文地址空间和系统级上下文地址空间一起构成了一个进程的整个存储器映像，如图 4.37 所示。

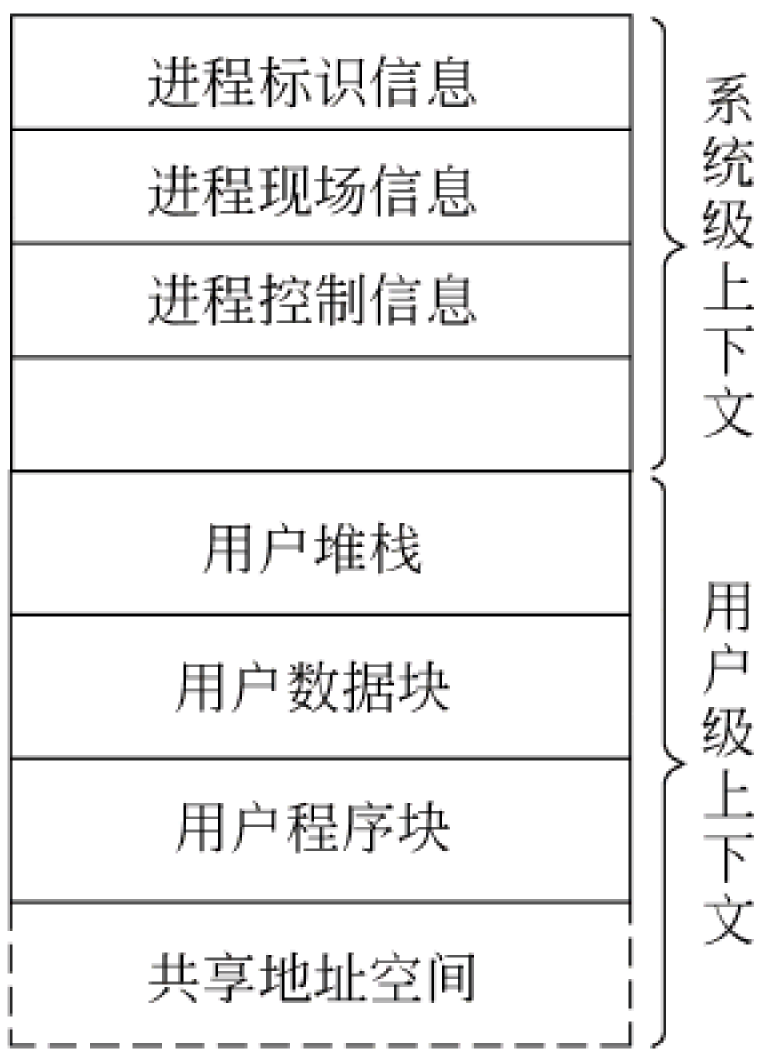


图 4.37　进程的存储映像

以下是经典的 hello.c 程序。

```
1  #include<stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6  }
```

对于上述高级语言源程序，首先需先对其进行预处理、编译成汇编语言表示，然后再用汇编程序将其转换为可重定位的二进制目标程序，再和库函数目标文件 printf.o 进行链接，生成最终的可执行目标文件 hello。

假定在 UNIX 系统上启动 hello 程序，其 shell 命令行和 hello 程序运行的结果如下。

```
unix> ./hello [Enter]
hello, world
unix>
```

上下文切换指把正在运行的进程换下，换一个新进程到处理器执行。图 4.38 给出了上述 shell 命令行执行过程中 shell 进程和 hello 进程的上下文切换过程：首先运行 shell 进程，从 shell 命令行中读入字符串“./hello”到主存；当 shell 进程读到字符“[Enter]”后，转到操作系统执行，由操作系统进行上下文切换，以保存 shell 进程的上下文并创建 hello 进程的上下文；hello 进程执行结束后，再转到操作系统完成将控制权从 hello 进程交回给 shell 进程。

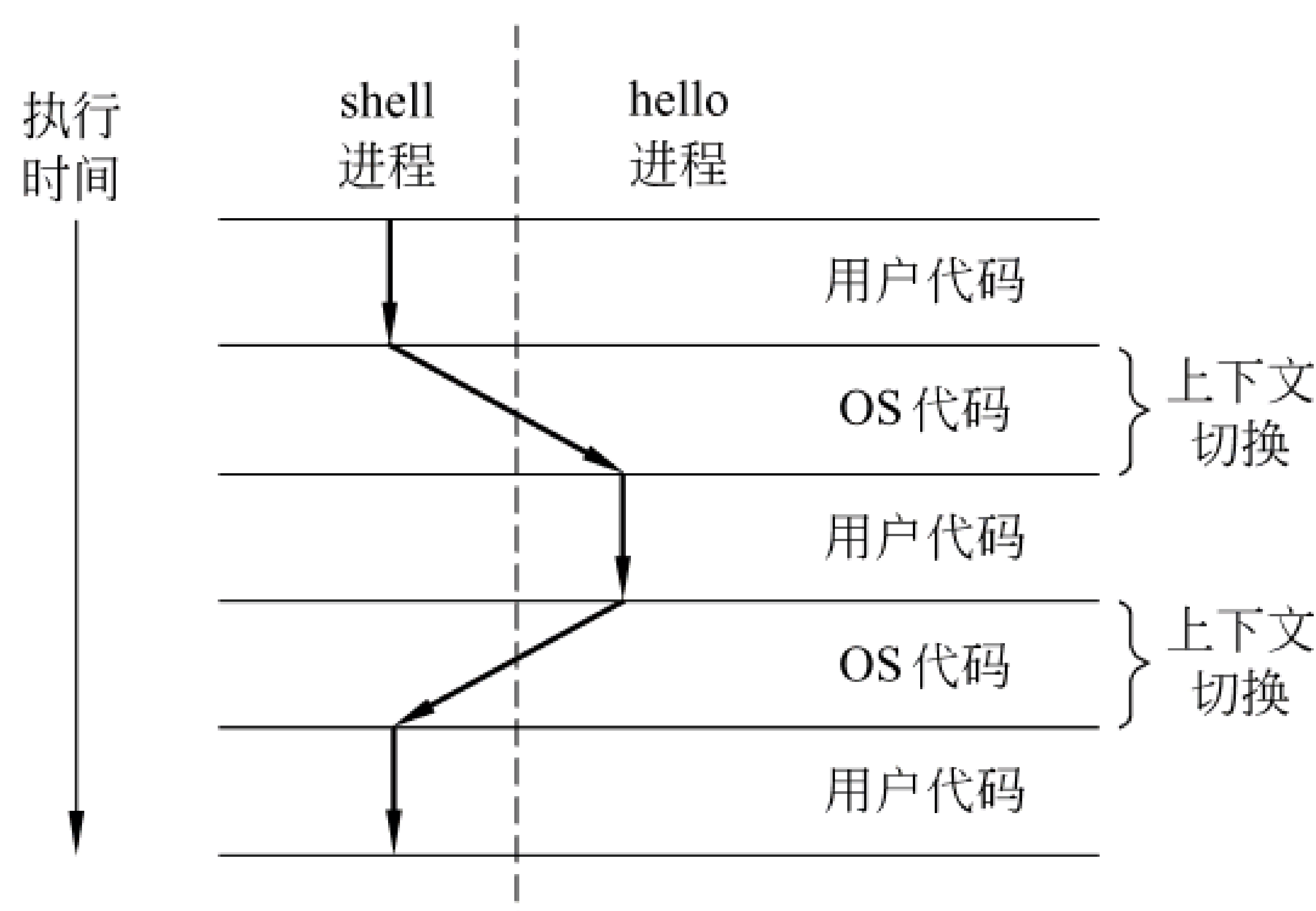


图 4.38　进程上下文切换示例

从上述过程可以看出，在一个进程的整个生命周期中，可能会有其他不同的进程在处理器中交替运行，所以，对于每个进程的运行时间很难准确、重复测量。这就是在第 1 章介绍性能评价时提到的性能测量问题的难点所在。

*4.7.2　存储器管理

早期计算机采用单道程序执行方式，系统的主存中仅包含操作系统（也称常驻监控程序）和正在执行的一个用户程序，所以无须进行存储管理，即使有也很简单。

现代计算机多采用多道程序执行方式，系统的主存中包含有操作系统和若干个用户程序。如果在主存中进程数很少，则当所有进程都需要等待 I/O 时，处理器就处于空闲状态。因此，需要对主存进行合理分配，尽可能让更多的进程进入主存，以便最大限度地利用处理器资源。

在多道程序系统中，主存储器的“用户”区需进一步划分给多个进程。划分的任务由 OS 动态执行，被称为存储器管理（memory management）。

早期经常采用交换（exchange）技术来使系统中尽量多地调入用户程序。其基本思想是：当主存中没有处于就绪状态的进程（例如，某一时刻所有进程都在等待 I/O）时，OS 将一些进程调出写回到磁盘，然后再调入其他进程来执行。

分区（partitioning）和分页（paging）是交换技术的两种实现方式。

1. 分区方式

分区方式将主存分为两大区域：系统区固定在一个地址范围内，存放操作系统；用户区用于存放所有用户程序。对于用户区的分配有简单固定分区(fixed-size partition)和可变长分区(variable-length partition)两种方式。

简单固定分区的基本思想为：使用长度不等的固定长分区，当一个进程调入主存时，分配一个能容纳它的最小分区给它。例如，在图4.39(a)所示的情况下，对于一个需要196K的进程，则将分区256K分配给它。多数情况下，进程对分区大小的需求不可能和提供的分区大小一样，因而，采用固定长度分区会大大浪费主存空间。

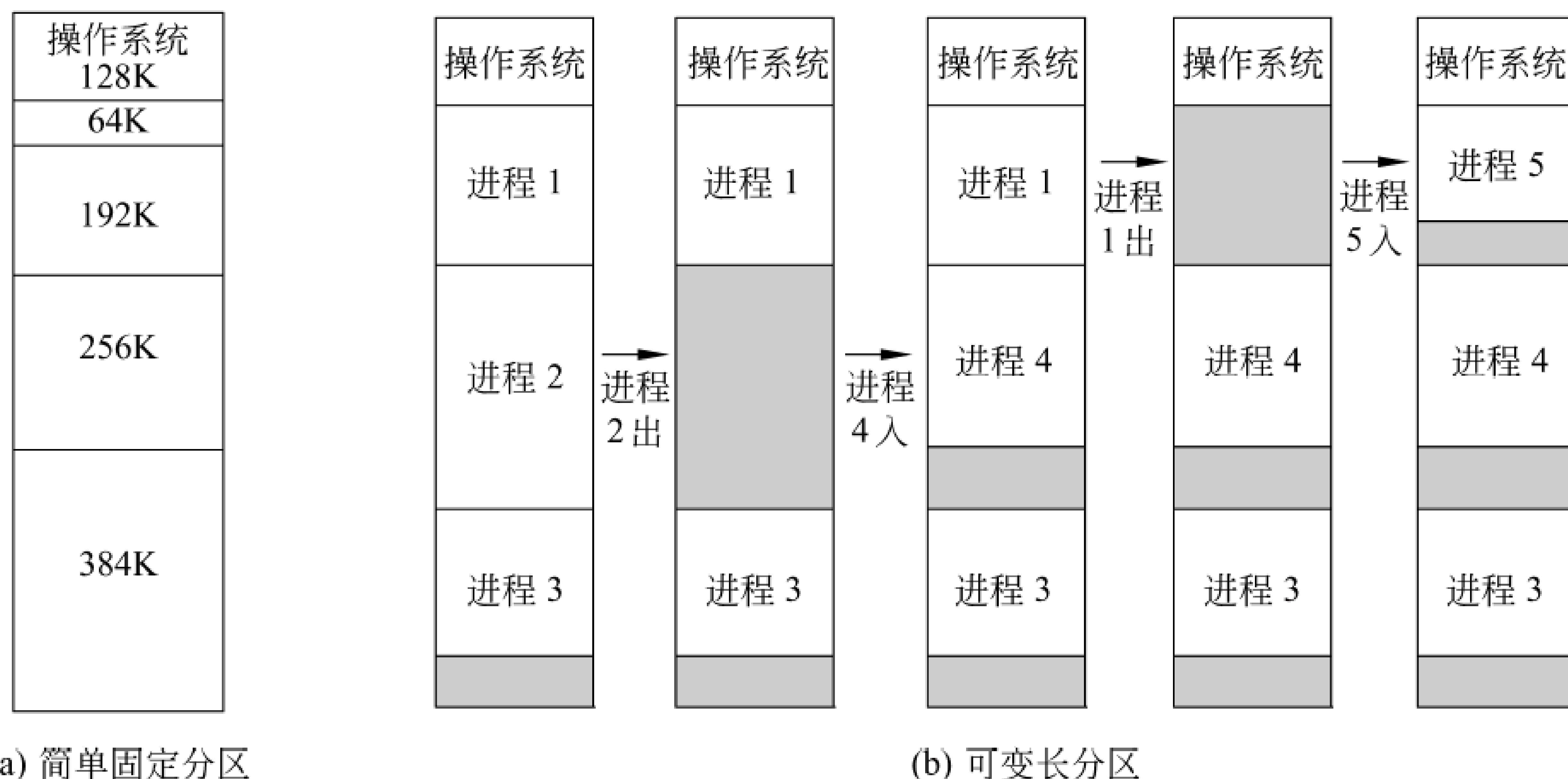


图4.39 分区存储管理方式

比简单固定分区更有效的方式是动态可变长分区，其分配的分区大小与进程所需大小一样。例如，在图4.39(b)所示的情况下，初始时，主存除了操作系统占据的空间外，其余空闲；把3个进程调入主存，假定从操作系统末端开始装入，结果在存储器末端留下一个很小的“空”块。因为它很小，不能装入第4个进程。因此，将进程2交换出来，此时，空间足够分配给进程4，但由于进程4的长度小于进程2的长度，分配后又产生了另一块“空”块。

由此可见，可变长分区方式开始时情况较好，但最后在存储器中可能会有许多小空块出现。时间越长，存储器中的碎片就会越多，因而存储器的利用率下降。

不管是固定分区还是可变长分区，都会在主存形成不能被利用的“碎片”。固定分区方式在分区内产生“内碎片”，而可变长分区方式在分区之间产生“外碎片”。通过移动进程将“碎片”合并可提高主存利用率，但会带来处理器的额外时间开销，并且，进程移动时要进行重定位，增加了重定位硬件开销。因此，分区方式不是解决多道程序运行的有效办法，现代多任务操作系统已较少使用这种方法。

2. 分页方式

分页方式的基本思想是：把主存分成固定长且比较小的存储块，称为页框(page frame)，每个进程也被划分成固定长的程序块，称为页(page)，程序块被装到可用的存储块中，并且无须用连续页框来存放一个进程。程序运行时装入内存页框的过程对程序员是透明的，因为程序员不需要知道程序运行时具体会装到哪些页框，因而程序给出的指令和数据

的地址不是真正的主存物理地址，程序员是在一个虚拟的逻辑地址空间中编写程序，通常把程序中所用的地址称为虚拟地址(virtual address)或逻辑地址(logical address)，而真正访问的主存地址称为物理地址(physical address)或实地址。操作系统在进行存储器分配时，通过页表(page table)建立页和页框之间的映射关系，每个进程有一个页表，通过页表实现虚拟地址向物理地址的转换。

早期的分页方式将一个进程的所有页面都调入主存，只是不像分区方式那样占用一块连续的物理内存，所以对主存的利用率比分区方式好，浪费的空间最多是最后一页的一部分。但是，它并没有很好地利用程序访问的局部性特点。根据程序访问的局部性可知，在一个进程的所有页面中，只有当前所访问的那个单元所在页面及邻近页面才是最近经常要访问的，早期分页方式将一个进程的所有页面都装入主存，使有限的页框资源被一些不活跃的页占用了，因而浪费了主存。为此，现代操作系统中采用了“请求分页(demand paging)”的分页式虚拟存储管理方式，所谓“请求分页”就是只将当前需要的页面装入主存页框中，而不需要的页面则存放在外存中。这就是现代计算机采用的“虚拟存储器(virtual memory)”存储管理的基本思想。

4.7.3 虚拟地址空间

虚拟存储器管理方式采用“请求分页”思想，每次访问仅将当前需要的页面调入主存，而进程中其他不活跃的页面放在外存磁盘上。当访问某个信息所在页不在主存时发生缺页，此时，从磁盘将缺失页面调入主存。

虚拟存储器机制为程序员提供了一个极大的虚拟(逻辑)地址空间，它是主存和磁盘I/O设备的抽象。虚存机制给每个进程带来了一个假象，好像每个进程都独占使用主存，并且主存空间极大。它带来了三个好处：(1)每个进程具有一致的虚拟地址空间，从而可以简化存储管理；(2)它把主存看成是磁盘存储器的一个缓存，在主存中仅保存活动的程序段和数据区，并根据需要在磁盘和主存之间进行信息交换，通过这种方式，使有限的主存空间得到了有效利用；(3)每个进程的虚拟地址空间是私有的，因此，可以保护各自进程不被其他进程破坏。

虚拟存储器机制中，每个源程序经编译、汇编、链接等处理生成可执行的二进制机器目标代码时，每个程序的目标代码都被映射到同样的虚拟地址空间，因此，所有用户进程的虚拟地址空间是一致的。

例如，图4.40给出了在Linux操作系统下hello程序的一个进程对应的虚拟地址空间映像。它分为两大部分：内核区(kernel area)和用户区(user area)。

内核区在0xC0000000以上的高端地址上，用来存放操作系统内核代码和数据以及与每个进程相关的数据结构(如进程标识信息、进程现场信息、页表等进程控制信息以及内核栈等)，其中内核代码和数据区在每个进程的地址空间中都相同。用户程序没有权限访问内核区。

用户区用来存放用户进程的代码和数据，它又被分为以下几个区域。

(1) 用户栈(user stack)。用来存放程序运行时过程调用的参数、返回值、返回地址、过程局部变量等，随着程序的执行，该区会不断动态地从高地址向低地址增长或向反方向减退(参见第5.4.4节)。

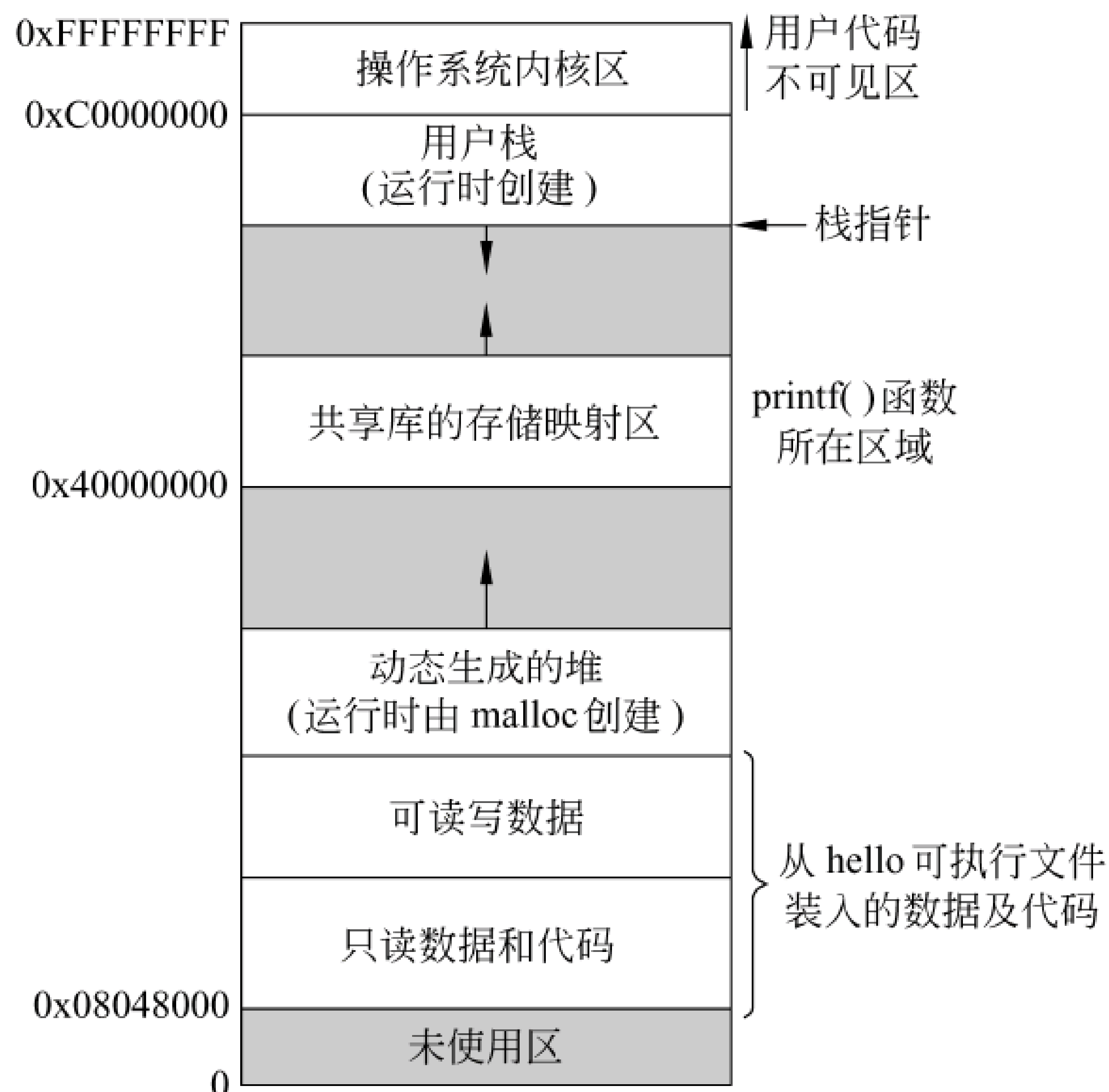


图 4.40 Linux 虚拟地址空间

(2) 共享库(shared libraries)。用来存放公共的共享函数库代码,如 hello 中的 printf()函数等。

(3) 堆(heap)。用于动态申请存储区,例如,C 语言中用 malloc()函数分配的变量区。申请一块内存时,动态地从低地址向高地址增长,用 free()函数释放一块内存时,动态地从高地址向低地址减退。

(4) 读写数据区。存放用户进程中的静态全局变量,堆区从该区域的结尾处开始向高地址增长。

(5) 只读代码/数据区。存放用户进程中的代码和只读数据,如 hello 进程中的程序代码和字符串"hello, world\n"。

每个区域都有相应的起始位置,堆区和栈区相向生长,栈区从内核起始位置 0xC0000000 开始向低地址增长,堆区中的共享库代码区从 0x40000000 开始向高地址增长。代码和只读数据区从 0x08048000 开始向高地址增长。

为了便于对存储空间的管理和存储保护,在规划存储映像时,通常将系统内核和用户进程分配在两端不同的区域。在用户进程区又把动态区和静态区分在两端,动态区中又把过程调用时的动态局部信息(栈区)和动态分配的内存区(堆区)分在两端,静态区中又把可读写区和只读区分在两端。这样的存储映像,便于每个区域的访问权限设置,因而便于存储保护和存储管理。

从图 4.40 可以看出,一个进程的虚拟地址空间中有一些"空洞"。例如,堆区和栈区都是动态生长的,因而在栈和共享库映射区之间、堆和共享库映射区之间都可能没有内容存在,这些没有和任何内容相关联的页称为"未分配页";对于代码和数据等有内容的区域所关联的页面,称为"已分配页"。已分配页中有两类:已调入主存而被缓存在 DRAM 中的页面称为"缓存页";未调入主存而存在磁盘上的页被称为"未缓存页"。因此,任何时刻一个进程中的所有

页面都被划分成三个不相交的页面集合：未分配页集合、缓存页集合和未缓存页集合。

4.7.4 虚拟存储器的实现

对照前面介绍的cache机制，可以把DRAM构成的主存看成是磁盘存储器的缓存。因此，要实现虚拟存储器机制，也必须考虑交换块（即页面）的大小问题、映射问题、写一致性问题等。

在cache（SRAM）中缓存的是主存块，而在主存（DRAM）中缓存的是虚拟页面，也就是磁盘中的程序块。DRAM比SRAM大约慢10倍，而磁盘比DRAM大约慢100 000多倍，因此，进行缺页处理所花的代价要比cache缺失损失大得多。而且，根据磁盘的特性，磁盘扇区定位所花的时间要比磁盘读写一个数据的时间长大约100 000倍，也即对扇区第一个数据的读写比随后数据的读写要慢100 000倍。考虑到缺页代价的巨大和磁盘访问第一个数据的开销，通常将主存和磁盘之间的交换页面设定得比较大，比在cache和主存之间交换的主存块大得多，典型的有4KB和8KB等。

由于缺页处理代价较大，所以应该尽量增加命中率，因此，在主存页框和虚拟页之间应该采用全相联映射方式。当进行写操作时，由于磁盘访问速度很慢，所以，不能每次写操作都同时写DRAM缓存和磁盘，也即应该采用回写（write back）方式，而不能用全写（write through）方式。

因为在虚拟存储机制中采用全相联映射，所以每个虚拟页可以存放到主存的任何一个空闲位置。因此，与cache一样，虚拟存储器机制必须要有一种方法来确定每个进程的各个页面所存放的对应主存位置或磁盘位置。根据对此问题解决方法的不同，虚拟存储器分成三种不同类型：分页式、分段式和段页式。

1. 分页式虚拟存储器

在分页式虚拟存储系统中，主存储器和虚拟地址空间都被划分成大小相等的页面，磁盘和主存之间按页面为单位交换信息。通常把虚拟地址空间中的页面称为虚拟页、逻辑页或虚页；主存空间中的页面被称为页框（页帧）、物理页或实页。有时虚拟页简称为VP（virtual page），物理页简称为PF（page frame）或PP（physical page）。对于这些概念的名称，不同教材的说法可能不同，但含义是一样的。

（1）页表（Page Table）

为了对每个虚拟页的存放位置、存取权限、使用情况、修改情况等进行说明，操作系统在主存中给每个进程都生成了一个页表，页表中对应每个虚拟页有一个表项，表项内容包含存放位置字段、装入位（valid）、修改位（dirty）、替换控制位、存取权限位和禁止缓存位等。

页表项中的存放位置字段用来建立虚拟页和物理页之间的映射，用于进行虚拟地址到物理地址的转换；装入位也称为有效位或存在位，用来表示对应页面是否在主存，若为“1”，表示该虚拟页已从外存调入主存，是一个“缓存页”，此时，位置字段指向主存页框号（即物理页号或实页号）；若为“0”，则表示没有被调入主存，此时，若位置字段为null，则说明是一个“未分配页”，否则是一个“未缓存页”，其位置字段给出该虚拟页在磁盘上的起始地址；修改位用来说明页面是否被修改过，虚存机制中采用回写（write back）策略，利用修改位可判断替换时是否需写回磁盘；替换控制位用来说明页面的使用情况，配合替换策略来设置，例如，是否最先调入（FIFO位），是否最近最少用（LRU位）等；访问权限位用来说明页面是可读可

写、只读还是只可执行等，用于存储保护；禁止缓存位用来说明页面是否可以装入 cache，通过正确设置该位，可以保证磁盘、主存和 cache 数据的一致性。图 4.41 给出了一个页表示例，其中，有 4 个缓存页：VP1、VP2、VP5 和 VP7；两个未分配页：VP0 和 VP4；两个未缓存页：VP3 和 VP6。

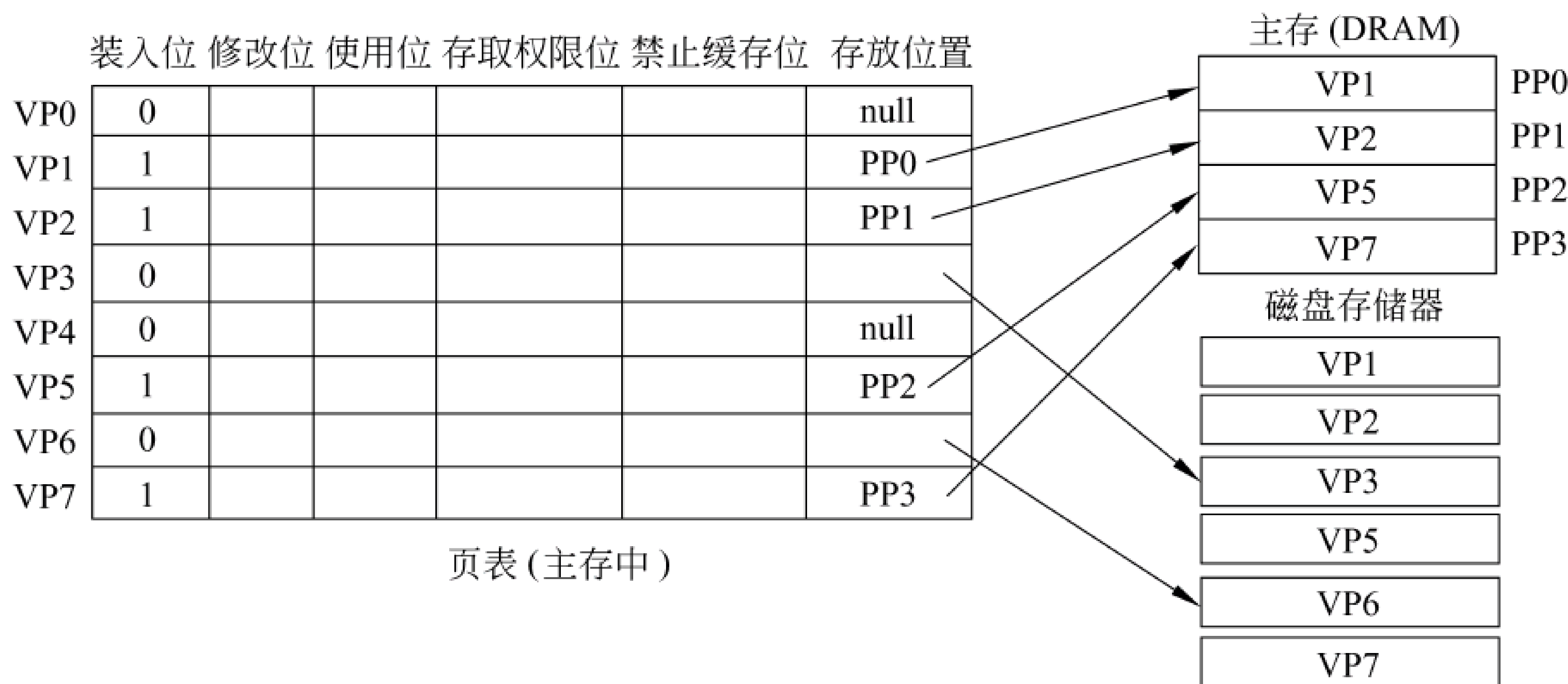


图 4.41 主存中的页表示例

对于图 4.41 所示的页表，假如 CPU 执行一条指令要求访问某个数据，若该数据正好在虚拟页 VP1 中，则根据页表得知，VP1 对应的装入位为 1，该页的信息存放在物理页 PP0 中，因此，可通过地址转换部件将虚拟地址转换为物理地址，然后到 PP0 中访问该数据；若该数据在 VP6 中，则根据页表得知，VP6 对应的装入位为 0，表示页面缺失，发生“缺页”异常，需要调出操作系统的“缺页”异常处理程序进行处理。“缺页”异常处理程序根据页表中 VP6 对应表项的存放位置字段，从磁盘中将所缺失的页面读出，然后找一个空闲的物理页框存放该页信息。若主存中没有空闲的页框，则还要选择一个页面替换到磁盘上。因为采用写回(write back)策略，所以页面淘汰时，需根据修改位确定是否要写回磁盘。缺页处理过程中需要对页表进行相应的更新。缺页异常处理结束后，程序回到原来发生缺页的指令继续执行。

对于图 4.41 所示的页表，虚拟页 VP0 和 VP4 是未分配页，但随着进程的动态执行，可能会使这些未分配页中有了具体的数据。例如，当调用 malloc 函数时，使堆区增长，新增的堆区正好与 VP4 对应，则内核就在磁盘上分配一个存储空间给 VP4，用于存放新增堆区中的内容，同时，对应 VP4 的页表项中的存放位置字段被填上该磁盘空间的起始地址，VP4 从未分配页转变为未缓存页。

系统中每个进程都有一个页表，页表属于进程控制信息，存放在进程地址空间的内核区，页表在主存的首地址记录在页表基址寄存器中。页表的项数由虚拟地址空间大小决定，前面提到，虚拟地址空间是一个用户编程不受其限制的足够大的地址空间。因此，页表项数会很多，因而带来页表过大的问题。例如：在 Intel x86 系统中，虚拟地址为 32 位，页面大小为 4KB，因此，一个进程有 $2^{32}/2^{12}=2^{20}$ 个页面，也即每个进程的页表可达 2^{20} 个页表项。若每个页表项占 32 位，则一个页表的大小为 4MB。显然，这么大的页表全部放在主存中是不适合的。

解决页表过大的方法有很多，可以采用限制大小的一级页表、二级或多级页表、倒置页表等方案。如何实现主要是操作系统考虑的问题，在此不多赘述。

（2）地址转换（Address Translationg）

对采用虚存机制的存储区进行的访问，指令中给出的地址是虚拟地址，所以，CPU 执行指令时，首先要将虚拟地址转换为主存物理地址，才能到主存取指令和数据。地址转换工作由 CPU 中的存储器管理部件（Memory Management Unit，MMU）来完成。

假设虚拟存储器中每个进程有 m 页，主存中有 n 个页框，通常情况下 $m>n$。由于页面大小是 2 的幂次，所以，每一页的起点都落在低位字段为零的地址上。因此，虚拟地址分为两个字段：高位字段为虚拟页号，低位字段为页内偏移地址。主存物理地址也分为两个字段：高位字段为物理页号，低位字段为页内偏移地址。由于两者的页面大小一样，所以页内地址是相等的。

分页式地址变换过程如图 4.42 所示。每个进程都有一个页表基址寄存器，存放该进程的页表首地址。首先根据页表基址寄存器的内容，找到对应的页表，然后由虚拟地址高位字段的虚拟页号为索引，找到对应的页表项，若装入位为 1，则取出物理页号，和虚拟地址中的页内地址拼接，形成实际主存物理地址；若装入位为 0，则说明缺页，需要操作系统进行缺页处理。

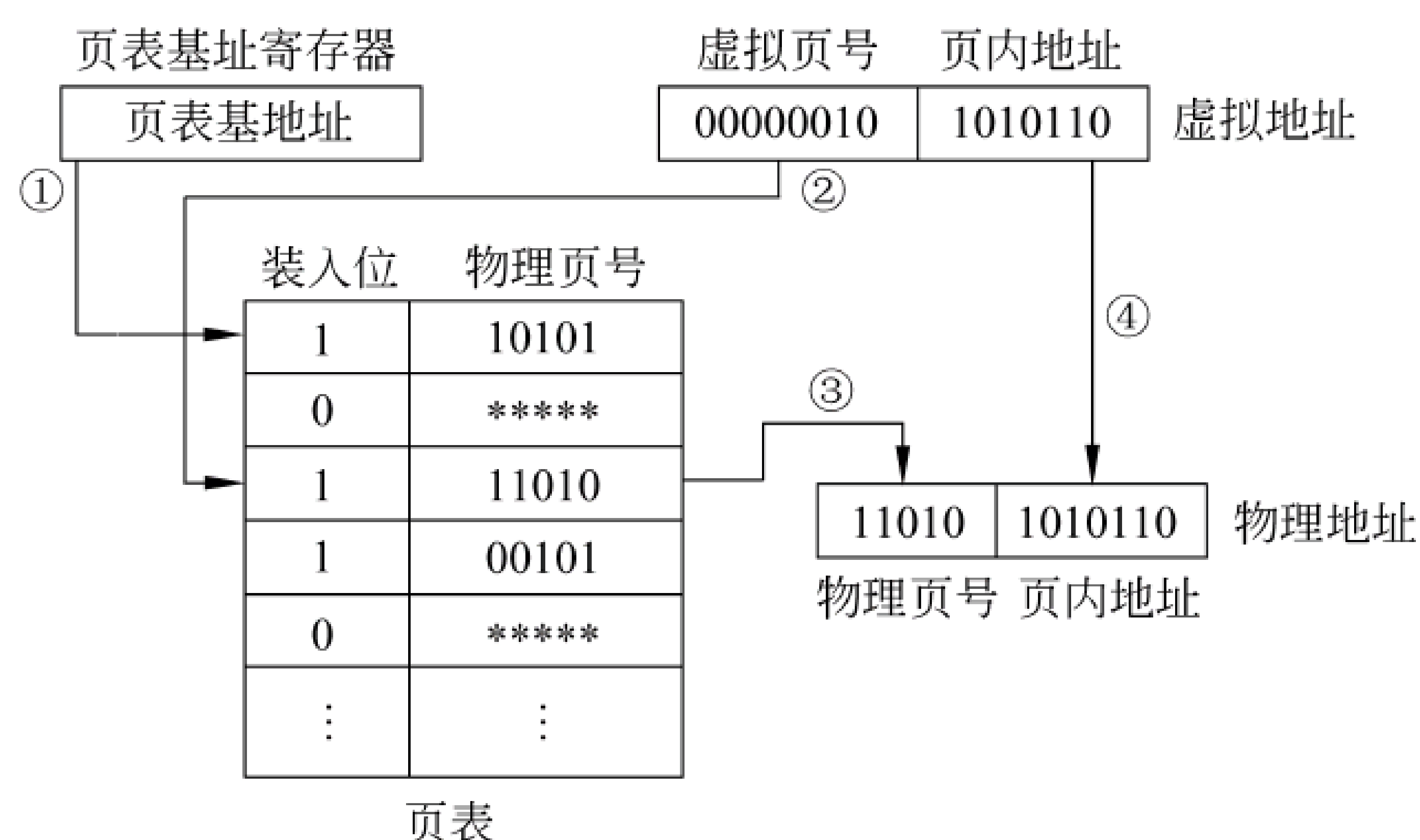


图 4.42　分页式虚存的地址转换

（3）快表（TLB）

从上述地址转换过程可看出，访存时首先要到主存查页表，然后才能根据主存物理地址再访问主存以存取指令或数据。如果缺页，则还要进行页面替换、页表修改等，访问主存的次数就更多。因此，采用虚拟存储器机制，使得访存次数增加了。为了减少访存次数，往往把页表中最活跃的几个页表项复制到高速缓存中，这种在高速缓存中的页表项组成的页表称为后备转换缓冲器（Translation Lookaside Buffer），通常简称为 TLB 或快表，相应地称主存中的页表为慢表。

这样，在地址转换时，首先到快表中查页表项，如果命中，则无须访问主存中的页表。因此，快表是减少访存时间开销的有效方法。

快表比页表小得多，为提高命中率，快表通常具有较高的关联度，大多采用全相联或组相联方式。每个表项的内容由页表表项内容加上一个 TLB 标记字段组成，TLB 标记字段用来表示该表项取自页表中的哪个虚拟页对应的页表项，因此，TLB 标记字段的内容在全相联方式下就是该页表项对应的虚拟页号；组相联方式下则是对应虚拟页号中的高位部分，而虚拟页号的低位部分是用于选择 TLB 组的组索引。

图4.43是一个具有TLB和cache的多级存储系统示意图，图中TLB和cache都采用组相联映射方式。

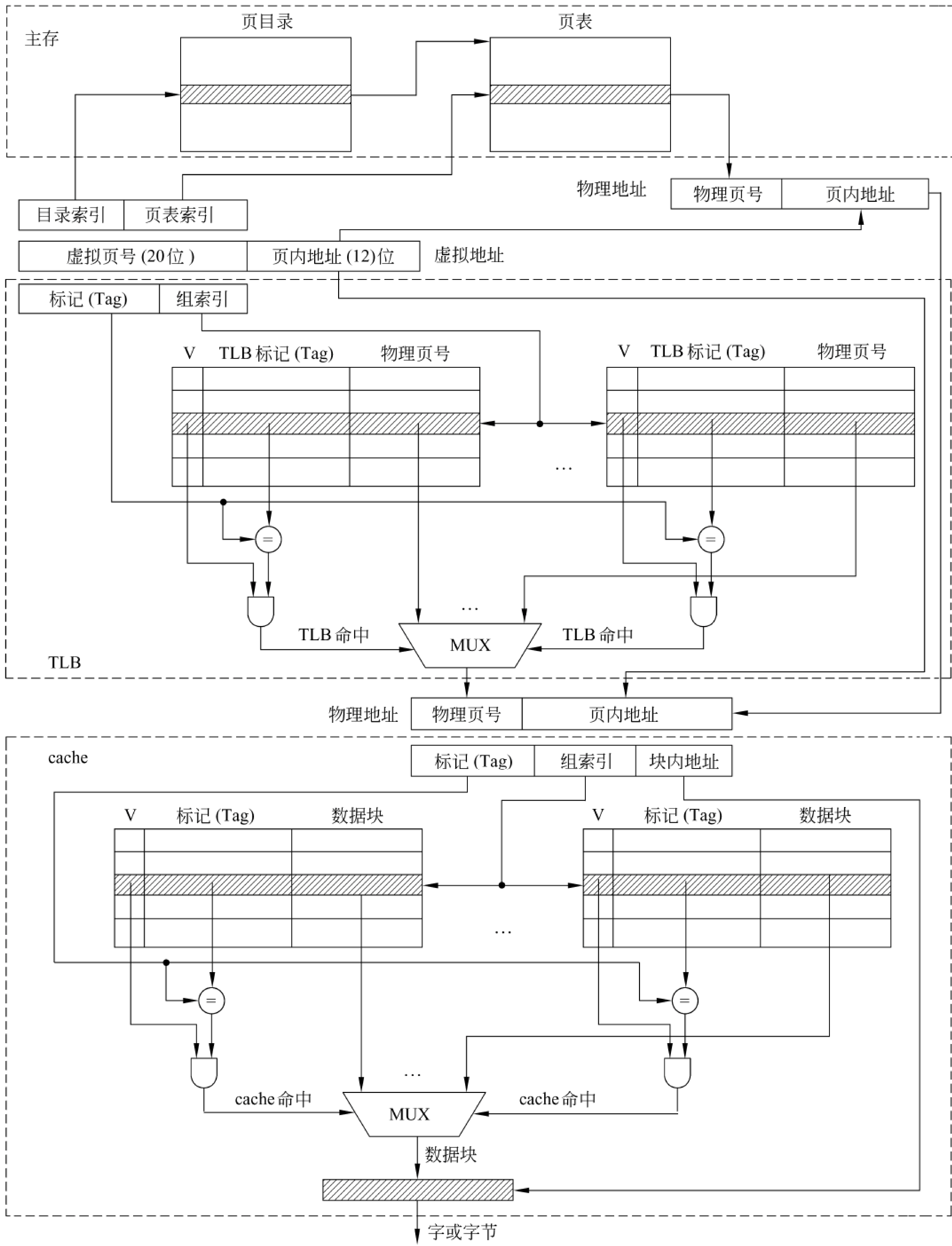


图4.43　TLB和cache的访问过程

在图 4.43 中，CPU 给出的是一个 32 位的虚拟地址，首先由 CPU 中的 MMU 进行虚拟地址到物理地址的转换；然后由处理 cache 的硬件根据物理地址进行存储访问。MMU 对 TLB 查表时，20 位的虚拟页号被分成标记（Tag）和组索引两部分，首先由组索引确定在 TLB 的哪一组进行查找。查找时将虚拟页号的标记部分与该组 TLB 中的每个 TLB 标记字段同时进行比较，若有某个相等且对应有效位为 1，则 TLB 命中，此时，可直接通过 TLB 进行地址转换；若都不相等，则 TLB 缺失，此时，需要访问主存去查页表，图中所示的是两级页表方式，虚拟页号被分成目录索引和页表索引两部分，由这两部分得到对应的页表项，从而进行地址转换，并将虚拟页号的高位部分作为 TLB 标记和对应页表项的内容一起送入 TLB。若 TLB 已满，还要进行 TLB 替换，为降低替换算法开销，TLB 常采用随机替换策略。在 MMU 完成地址转换后，cache 硬件根据映射方式将转换得到的主存物理地址划分成多个字段，然后，根据 cache 索引，找到对应的 cache 行或 cache 组，将对应各 cache 行中的标记与物理地址中的高位地址进行比较，若相等且有效位为 1，则 cache 命中，此时，根据块内地址取出对应的字，需要的话，再根据字节偏移量从字中取出相应字节送 CPU。

目前 TLB 的一些典型指标为：TLB 大小为 16～512 项，块大小为 1～2 项（每个表项 4～8B），命中时间为 0.5～1 个时钟周期，缺失损失为 10～100 个时钟周期，命中率为 90%～99%。

（4）CPU 访存过程

在一个具有 cache 和虚拟存储器的系统中，CPU 的一次访存操作可能涉及到 TLB、页表、cache、主存和磁盘的访问，其访问过程如图 4.44 所示。

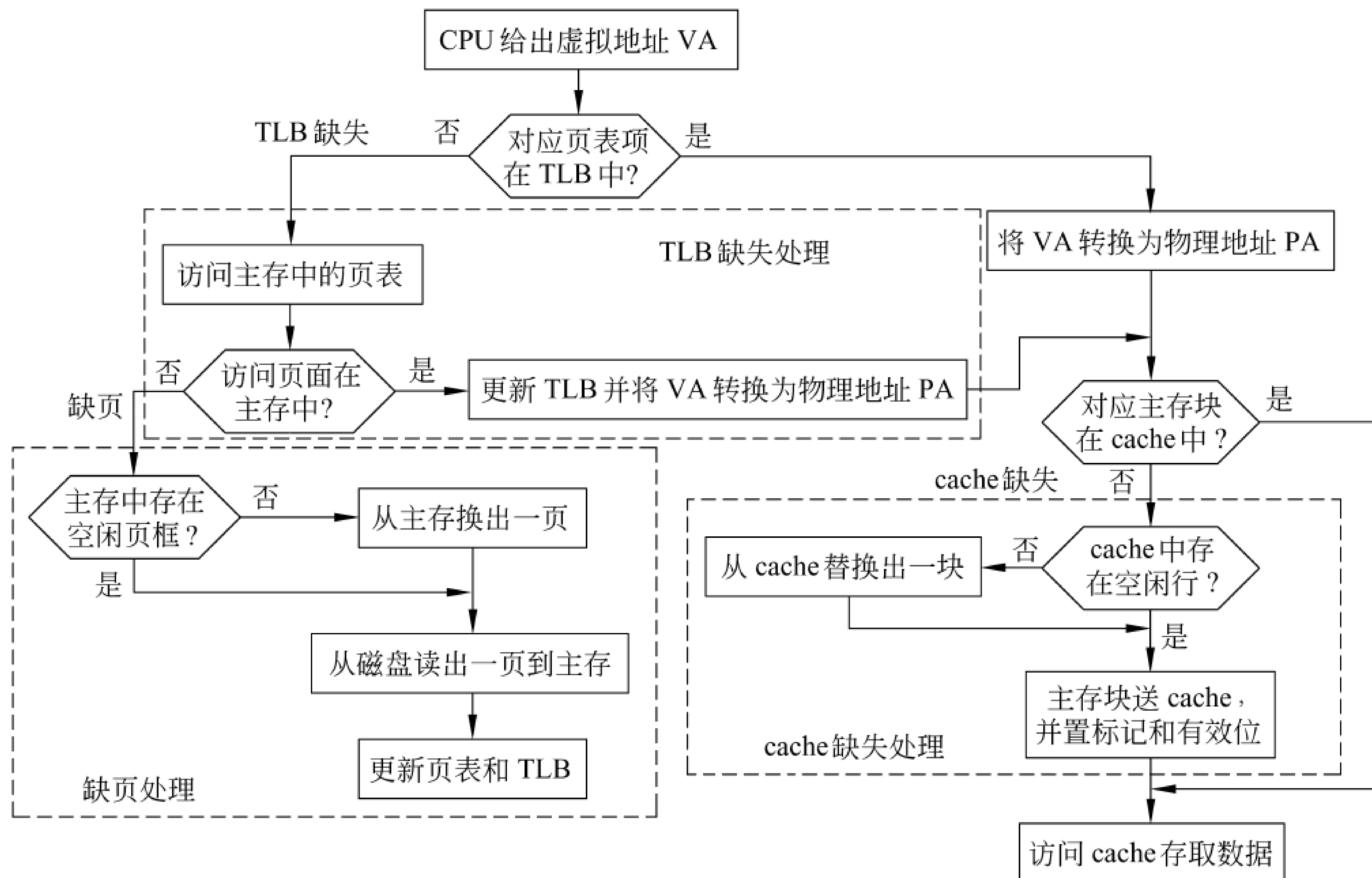


图 4.44　CPU 访存过程

从图 4.44 可以看出，CPU 访存过程中存在以下三种缺失情况。

(1) TLB 缺失(TLB miss):要访问的页面对应的页表项不在 TLB 中。

(2) cache 缺失(cache miss):要访问的主存块不在 cache 中。

(3) 缺页(page miss):要访问的页面不在主存中。

表 4.1 给出了三种缺失的几种组合情况。

表 4.1　TLB、page、cache 三种缺失组合

序号	TLB	page	cache	说　明
1	hit	hit	hit	可能,TLB 命中则页一定命中,信息在主存,就可能在 cache 中
2	hit	hit	miss	可能,TLB 命中则页一定命中,信息在主存,但可能不在 cache 中
3	miss	hit	hit	可能,TLB 缺失但页可能命中,信息在主存,就可能在 cache 中
4	miss	hit	miss	可能,TLB 缺失但页可能命中,信息在主存,但可能不在 cache 中
5	miss	miss	miss	可能,TLB 缺失,则页也可能缺失,信息不在主存,一定也不在 cache
6	hit	miss	miss	不可能,页缺失,说明信息不在主存,TLB 中一定没有该页表项
7	hit	miss	hit	不可能,页缺失,说明信息不在主存,TLB 中一定没有该页表项
8	miss	miss	hit	不可能,页缺失,说明信息不在主存,cache 中一定也没有该信息

很显然,最好的情况是第 1 种组合,此时,无须访问主存;第 2 和 3 两种组合,都需要访问一次主存;第 4 种组合要访问两次主存;而第 5 种组合,则发生"缺页"异常,需访问磁盘,并至少访存两次。

cache 缺失处理由硬件完成;缺页处理由软件完成,操作系统通过"缺页异常处理程序"来实现;而对于 TLB 缺失,则可以用硬件也可以用软件来处理。用软件方式处理时,操作系统通过专门的"TLB 缺失异常处理程序"来实现。

对于分页式虚拟存储器,其页面的起点和终点地址固定。因此,实现简单,开销少。而且因为只有进程的最后一个零头(内部碎片)不能利用,故浪费很小。但是,由于页不是逻辑上独立的实体,因此,对于那些不采用对齐方式存储的计算机来说,可能会出现一个数据或一条指令分跨在两个页面等问题,使处理、管理、保护和共享等都不方便。采用下面介绍的分段式虚拟存储器就可避免这种情况的发生。

2. 分段式虚拟存储器

根据程序的模块化性质,可按程序的逻辑结构划分成多个相对独立的部分,例如,过程、数据表、数据阵列等。这些相对独立的部分被称为段,它们作为独立的逻辑单位可以被其他程序段调用,形成段间连接,从而产生规模较大的程序。段通常有段名、段起点、段长等。段名可用用户名、数据结构名或段号标识,以便于程序的编写、编译器的优化和操作系统的调度管理等。

可以把段作为基本信息单位在主存和辅存之间传送和定位。分段方式下,将主存空间按实际程序中的段来划分,每个段在主存中的位置记录在段表中,段的长度可变,所以段表中需有长度指示。每个进程有一个段表,每个段在段表中有一个段表项,用来指明对应段在主存中的位置、段长、访问权限、使用和装入情况等。段表本身也是一个可再定位段,可以存在外存中,需要时调入主存,但一般驻留在主存中。

在分段式虚拟存储器系统中，虚拟地址由段号和段内地址组成。通过段表把虚拟地址变换成主存物理地址，其变换过程如图 4.45 所示。

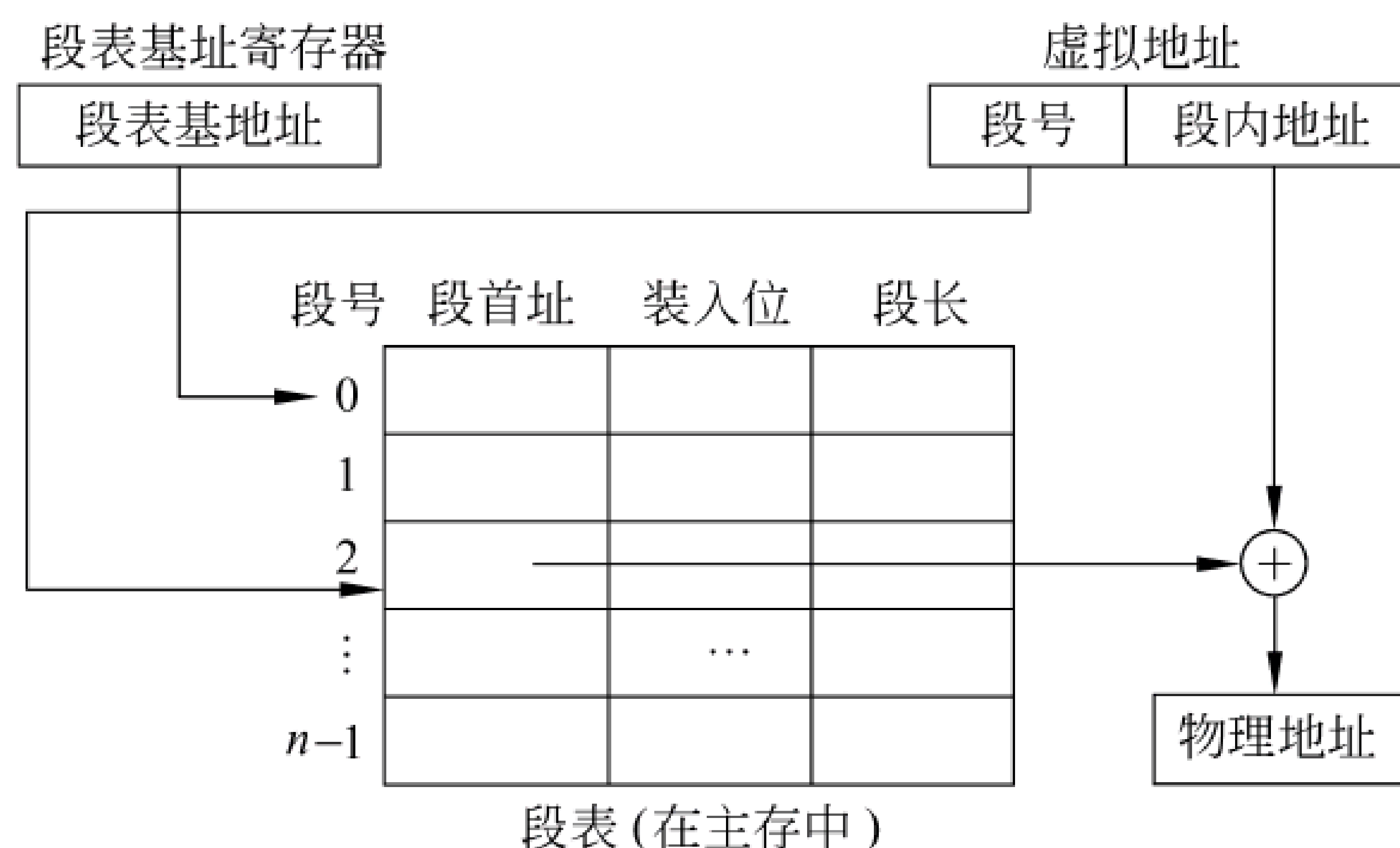


图 4.45 分段式虚存的地址转换

每个用户进程有一个段表基址寄存器，存放其段表在内存的首地址。根据虚拟地址中的段号，找到对应段表项，检查是否存在以下三种异常情况。

(1) 缺段(段不存在)：装入位＝0。

(2) 地址越界：偏移量超出最大段长。

(3) 保护违例：操作方式与指定访问权限不符。

若发生以上三种情况，则调用相应的异常处理程序，否则，将段表项中的段首址与虚拟地址中的段内地址相加，生成主存物理地址。

因为段本身是程序的逻辑结构所决定的一些独立部分，因而分段对程序员来说是不透明的；而分页对程序员来说是透明的，程序员编写程序时，不需知道程序将如何分页。

分段式管理系统的优点是段的分界与程序的自然分界相对应；段的逻辑独立性使它易于编译、管理、修改和保护，也便于多道程序共享；某些类型的段(如堆、栈、队列等)具有动态可变长度，允许自由调度以便有效利用主存空间。但是，由于段的长度各不相同，段的起点和终点不固定，给主存空间分配带来麻烦，而且容易在主存中留下许多空白的零碎空间，造成浪费。

分段式和分页式存储管理各有优缺点，因此可采用分段和分页相结合的段页式存储管理方式。

3. 段页式虚拟存储器

在段页式虚拟存储器中，程序按模块分段，段内再分页，用段表和页表(每段一个页表)进行两级定位管理。段表中每个表项对应一个段。每个段表项中包含一个指向该段页表起始位置的指针以及该段其他的控制和存储保护信息，由页表指明该段各页在主存中的位置以及是否装入、修改等状态信息。

程序的调入调出按页进行，但它又可以按段实现共享和保护。因此，它兼有页式和段式的优点。它的缺点是在地址映像过程中需要多次查表。

多道程序中的每个用户进程需要一个基号，用于标识用户进程，进程的段表起始地址存放在各自对应的基址寄存器中。这样，虚拟地址由基号、段号、页号和页内地址组成。格式如下。

基号	段号	页号	页内地址

例如,假定有三个进程,基号分别为1、2、3,其基址寄存器内容分别为B1,B2,B3,逻辑地址到物理地址的变换过程如图4.46所示。

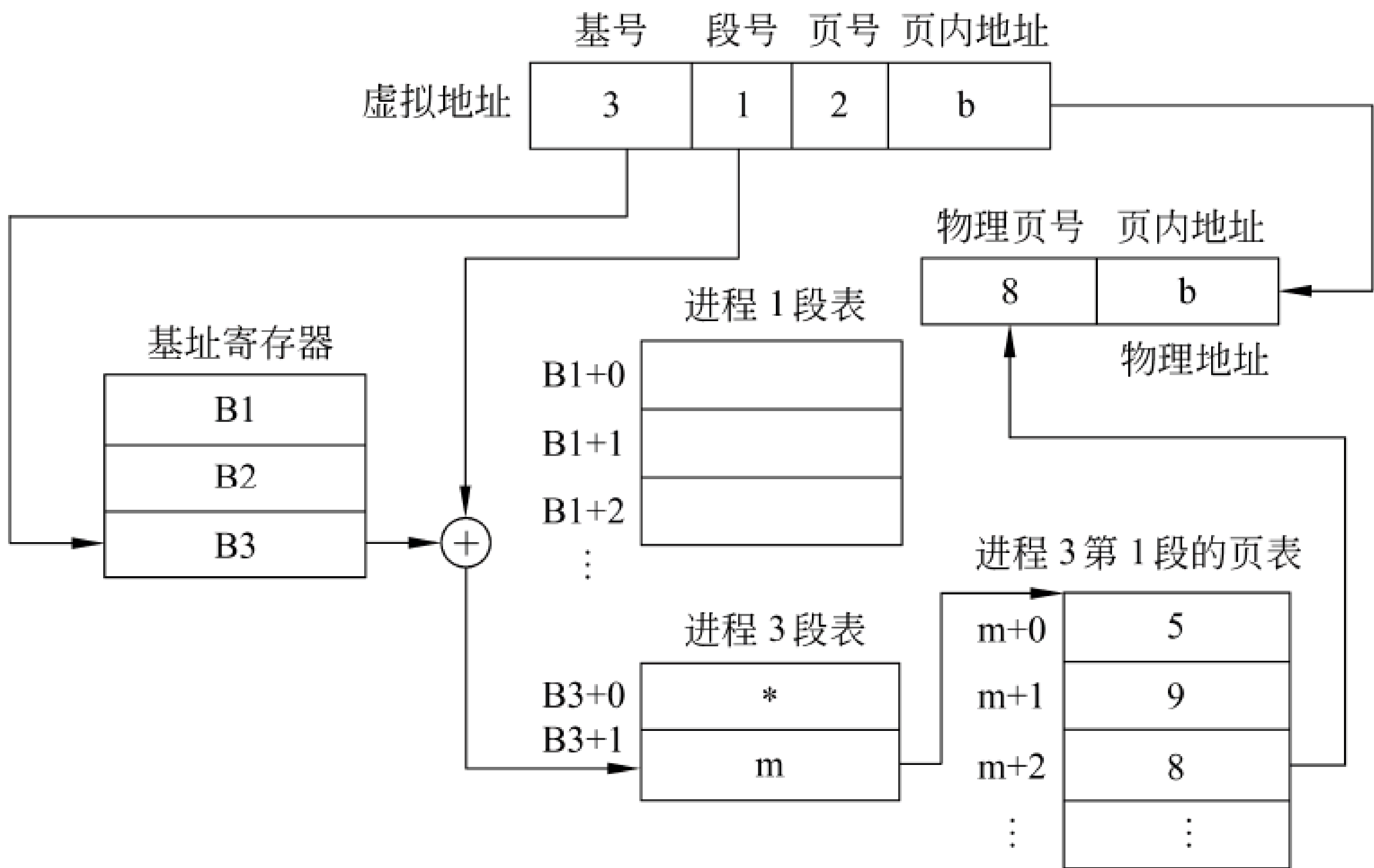


图4.46　段页式虚存的地址转换

段页式地址转换过程如下:

(1) 根据基号找到第3个基址寄存器,将该基址寄存器的内容B3和段号1相加,得到进程3的段表项地址,其内容为对应页表的起始地址m。

(2) 根据虚拟地址中的页号2,在起始地址为m的页表中找到第2个页表项,其内容为物理页号8。

(3) 将物理页号与页内地址拼接,得到物理地址。

图4.46中的段表和页表都是简化形式,表中只给出了用于地址转换的一些位置信息,而省略了其他各种控制信息,实际上在段表中应该还有访问权限、段长等信息,页表中还有装入位、修改位等其他信息。与分页式和段页式虚存一样,在地址转换过程中,也要判断是否发生了地址越界、访问越权或缺页等异常情况。

*4.7.5　存储保护

为避免主存中多道程序相互干扰,防止某程序出错而破坏其他程序的正确性,或某程序不合法地访问其他程序或数据区,应该对每个程序进行存储保护,包括操作系统程序和用户程序。

为了对操作系统存储保护提供支持,硬件必须具有以下三种基本功能。

(1) 支持至少两种运行模式:管理模式(Supervisor Mode)和用户模式(User Mode)。

用于完成操作系统各种功能的进程为系统进程,也称为内核(Kernel)进程、管理(Supervisor)进程。执行系统进程时,处理器所处的模式称为管理模式,或称管理程序状态,简称管态、管理态或核心态。

完成非操作系统功能的进程称为用户进程,当运行用户进程时,处理器所处的模式就是用户模式,或称用户状态、目标程序状态,简称为目态或用户态。

(2) 使部分 CPU 状态只能由系统进程写而用户进程只能读不能写。

这部分状态包括 User/Supervisor 模式位、页表首地址、TLB 内容等。OS 内核可用特殊指令(一般称为管态指令)来写这些状态。

(3) 提供让 CPU 在管理模式和用户模式之间相互转换的机制。

通过"异常"处理和"陷阱"(系统调用)使 CPU 从用户模式转到管理模式;异常处理中的"返回"指令(return from exception)使 CPU 从管理模式转到用户模式。

硬件通过提供相应的专用寄存器、专门的指令、专门的状态/控制位等,和操作系统一起实现上述三个功能。通过这些功能,并把页表保存在操作系统的地址空间,操作系统就可更新页表,并防止用户进程改变页表,以确保用户进程只能访问由 OS 分配给的存储空间。

存储保护包括以下两种情况:访问方式保护和存储区域保护。

1. 访问方式保护

访问方式保护就是看是否发生了"访问越权"。若实际访问操作与访问权限不符则发生存储保护错。

通常通过在页表或段表中设置访问权限位来实现。一般规定:各道程序对本程序所在的存储区可读可写;对共享区或已获授权的其他用户信息可读不可写;而对未获授权的信息(如 OS 内核、页表等)不可访问。通常,数据段可指定为可读可写或只读;程序段可指定只可执行或只读。

2. 存储区域保护

存储区域保护就是看是否发生了"地址越界"。若访问了不该访问的区域,则发生存储保护错。通常有以下几种常用的存储区域保护方式。

(1) 加界重定位。每个程序或程序段都记录有起始地址和终止地址,分别称为上界和下界。对虚拟地址加界(即加基准地址)生成物理地址后,如果物理地址超过了上界和下界规定的范围,则地址越界。有些系统用专门的一对上界寄存器和下界寄存器来记录上界和下界,在分段式虚存中,通过段表来记录段的上界和下界。

(2) 键保护方式。操作系统为主存的每一个页框分配一个存储键,为每个用户进程设置一个程序键。进程运行时,将程序状态字寄存器中的键(程序键)和所访问页的键(存储键)进行核对,相符时才可访问,如同"锁"与"钥匙"的关系。为使某个页框能被所有进程访问,或某个进程可访问任何一个页框,可规定键标志为 0,此时不进行核对工作。例如,操作系统有权访问所有页框中的页面,因此,可让内核进程的程序键为 0。

(3) 环保护方式。主存中各进程按其重要性分为多个保护级,各级别构成同心环,最内环的进程保护级别最高,向外逐次降低。内环进程可以访问外环和同环进程的地址空间,而外环不得访问内环进程的地址空间。系统进程的保护级别最高,环号最小,而用户进程都处于外环上,Pentium 就采用该方案。

4.8 本章小结

本章主要内容包括存储器的分类、存储器的分层结构、半导体随机存取存储器的组织、只读存储器、主存储器与 CPU 的连接、多体交叉编址存储器、高速缓冲存储器 cache 的基本原理、cache 和主存之间的地址映射、替换算法、虚拟存储器的基本概念、页表结构、缺页异常、转换后援缓冲器 TLB 等。具体总结如下:

- 存储器的分类
  - 按存取方式分：随机、顺序、直接、相联。
  - 按存储介质分：半导体、磁表面、激光盘。
  - 按信息可更改性：可读可写、只读。
  - 按断电后可否保存：易失、非易失。
  - 按功能/容量/速度分：寄存器、cache、主存(内存)、辅存(外存)。
- 存储器的分层结构：按速度从快到慢、容量从小到大、价格从贵到便宜，以及与CPU连接的距离由近到远的顺序，构成的层次化存储结构为寄存器→cache→主存→磁盘→光盘、磁带。
- 半导体随机存取存储器的组织
  存储元(记忆单元)→存储芯片→存储模块(内存条)→存储器。
- 只读存储器：MROM、PROM、EPROM、EEPROM、Flash ROM。
- 存储器芯片与CPU的连接
  - 地址线的连接：考虑芯片在字方向上扩展，连续编址时低位用于芯片内地址、高位用于片选逻辑，被送到片选信号译码器，译码输出连到芯片的片选信号引脚上。
  - 数据线的连接：考虑芯片在位方向上扩展，分别连到位扩展的芯片上。
  - 控制线的连接：读写信号、主存/IO访问信号等经过组合连到芯片相应的引脚。
- 主存的主要技术指标
  - 存取时间：执行一次读操作或写操作的时间，分读出时间和写入时间。
  - 存储周期：存储器进行连续两次独立的读或写操作所需的最小时间间隔。
  - 存储器带宽：每秒钟从存储器进出信息的最大数量。
- 主存储器的校验：采用海明校验方式。
- 多模块存储器
  - 连续编址：按高位地址划分模块，地址在一个存储模块内连续编号。
  - 交叉编址：按低位地址划分模块，地址在所有存储模块之间交叉编号。
- 高速缓存(cache)
  - 基本原理：利用程序访问的局部性特点，把主存中的一块数据复制到cache。
    - 时间局部性：某单元在一个很短的时间段内可能被重复访问。
    - 空间局部性：某单元被访问后，其邻近单元不久也可能被访问。
  - cache和主存间的映射
    - 直接映射：每个主存块只能存放到一个固定的cache行中。
    - 全相联映射：每个主存块可以存放到任何一个cache行中。
    - 组相联映射：cache分若干组，每组有多行，各主存块存放到固定组的任意行中。
  - 替换算法
    - FIFO：总是把最先调到cache的那个主存块淘汰掉。
    - LRU：总是把最近最少用到的那个主存块淘汰掉。
  - 写策略
    - 写回法：暂时只写cache，替换时一次性写回主存。
    - 全写法：每次写cache时也写主存，可在cache和主存间加写缓存。

● 虚拟存储器

◆ 基本原理：每个进程具有一个一致的、极大的、私有的虚拟地址空间，虚拟地址空间按等长的页来划分，主存也按等长的页框划分。进程执行时将当前用到的页面装入主存，其他暂时不用的部分放在磁盘上，通过页表建立虚拟页和主存页框之间的对应关系。不在主存的页面在页表中记录其在磁盘上的地址。在指令执行过程中，由特殊的硬件(MMU)和操作系统一起实现存储访问。

◆ 虚拟存储器的实现方案：分页式、分段式、段页式。

◆ 地址转换：根据虚拟地址中的虚页号，找到对应的页表项。通过页表项得到对应虚页的页框号(即物理页号、实页号)，将它和页内地址拼接得到物理地址。

◆ 页表和页表项：每个进程有一个页表，每个页表项由有效(装入)位、使用位、修改位、存取权限位、主存页框号或磁盘地址等组成。

◆ "缺页"：地址转换过程中发现所需页面不在主存。操作系统的缺页处理程序从磁盘读入所需页面到主存，并修改页表。缺页处理后，须回到原来发生缺页的指令重新执行。

◆ TLB(快表)：用来存放常用页表项，以减少到主存访问页表的次数。

◆ 存储保护：有地址越界和访问越权两种存储保护错(即访问违例)。

习 题 4

1. 给出以下概念的解释说明。

(1) 静态 RAM(SRAM)　(2) 动态 RAM(DRAM)　(3) 刷新
(4) 易失性存储器　(5) 相联存储器　(6) 存取时间
(7) 存储周期　(8) 存储器带宽　(9) 片选信号
(10) 地址引脚复用　(11) 行选通信号(RAS)　(12) 列选通信号(CAS)
(13) PROM　(14) EPROM　(15) E^2PROM
(16) 闪存(Flash 存储器)　(17) 多模块交叉存储器　(18) 双口 RAM
(19) 时间局部性　(20) 空间局部性　(21) 命中率
(22) 命中时间　(23) 缺失率　(24) 缺失损失
(25) 虚拟地址(逻辑地址)　(26) 虚拟页号(虚页号)　(27) 物理地址(主存地址)
(28) 页框(页帧)　(29) 物理页号(实页号)　(30) 重定位
(31) 页表　(32) 页表基址寄存器　(33) 有效位(装入位)
(34) 修改位　(35) 缺页(page fault)　(36) 请求分页
(37) FIFO　(38) LRU　(39) 快表(TLB)
(40) 管理模式　(41) 用户模式　(42) 异常返回
(43) 存储保护　(44) 地址越界　(45) 访问越权

2. 简单回答下列问题。

(1) 计算机内部为何要采用层次化存储体系结构？层次化存储体系结构如何构成？
(2) SRAM 芯片和 DRAM 芯片各有哪些特点？各自用在哪些场合？
(3) CPU 和主存之间有哪两种通信定时方式？SDRAM 芯片采用什么方式和 CPU 交换信息？
(4) 为什么在 CPU 和主存之间引入 cache 能提高 CPU 访存效率？
(5) 为什么说 cache 对程序员来说是透明的？

(6) 什么是 cache 映射的关联度？关联度与命中率、命中时间的关系各是什么？

(7) 为什么直接映射方式不需要考虑替换策略？

(8) 为什么要考虑 cache 的一致性问题？读操作时是否要考虑 cache 的一致性问题？为什么？

(9) 什么是物理地址？什么是逻辑地址？地址转换由硬件还是软件实现？为什么？

(10) 什么是页表？什么是快表(TLB)？

(11) 在存储器层次化结构中，“cache—主存”、“主存—磁盘”这两个层次有哪些不同？

3. 已知某机主存容量为 64KB，按字节编址。假定用 1K×4 位的 DRAM 芯片构成该存储器，要求回答以下问题。

(1) 需要多少个这样的 DRAM 芯片？

(2) 主存地址共多少位？哪几位用于选片？哪几位用于片内选址？

(3) 画出该存储器的逻辑框图。

4. 假定用 64K×1 位的 DRAM 芯片构成 256K×8 位的存储器，要求回答以下问题。

(1) 所需芯片数为多少？画出该存储器的逻辑框图。

(2) 若采用异步刷新方式，每单元刷新间隔不超过 2ms，则产生刷新信号的间隔是多少时间？若采用集中刷新方式，则存储器刷新一遍最少用多少个读写周期？

5. 假定用 8K×8 位的 EPROM 芯片组成 32K×16 位的只读存储器，要求回答以下问题。

(1) 数据寄存器最少应有多少位？

(2) 地址寄存器最少应有多少位？

(3) 共需多少个 EPROM 芯片？

(4) 画出该只读存储器的逻辑框图。

6. 某计算机中已配有 0000H～7FFFH 的 ROM 区域，现在再用 8K×4 位的 RAM 芯片形成 32K×8 位的存储区域，CPU 地址线为 A0～A15，数据线为 D0～D7，控制信号为 $R/\overline{W}$(读写)、$\overline{MREQ}$(访存)。要求说明地址译码方案，并画出 ROM 芯片、RAM 芯片与 CPU 之间的连接图。假定上述其他条件不变，只是 CPU 地址线改为 24 根，地址范围 000000H～007FFFH 为 ROM 区，剩下的所有地址空间都用8K×4 位的 RAM 芯片配置，则需要多少个这样的 RAM 芯片？

7. 假定一个存储器系统支持四体交叉存取，某程序执行过程中访问地址序列为 3，9，17，2，51，37，13，4，8，41，67，10，则哪些地址访问会发生体冲突？

8. 现代计算机中，SRAM 一般用于实现快速小容量的 cache，而 DRAM 用于实现慢速大容量的主存。以前超级计算机通常不提供 cache，而是用 SRAM 来实现主存(如 Cray 巨型机)，请问：如果不考虑成本，你还这样设计高性能计算机吗？为什么？

9. 对于数据的访问，分别给出具有下列要求的程序或程序段的示例。

(1) 几乎没有时间局部性和空间局部性。

(2) 有很好的时间局部性，但几乎没有空间局部性。

(3) 有很好的空间局部性，但几乎没有时间局部性。

(4) 空间局部性和时间局部性都好。

10. 假定某计算机主存地址空间大小为 1GB，按字节编址，cache 的数据区(不包括标记、有效位等)有 64KB，块大小为 128 字节，采用直接映射和直写(write-through)方式。请回答以下问题。

(1) 主存地址如何划分？要求说明每个字段的含义、位数和在主存地址中的位置。

(2) cache 的总容量为多少位？

11. 假定某计算机的 cache 共 16 行，开始为空，块大小为一个字，采用直接映射方式，按字编址。CPU 执行某程序时，依次访问以下地址序列：2,3,11,16,21,13,64,48,19,11,3,22,4,27,6 和 11。要求：

(1) 说明每次访问是命中还是缺失，试计算访问上述地址序列的命中率。

(2) 若 cache 数据区容量不变，而块大小改为 4 个字，则上述地址序列的命中情况又如何？

12. 假定数组元素在主存按从左到右的下标顺序存放。试改变下列函数中循环的顺序，使得其数组元素的访问与排列顺序一致，并说明为什么修改后的程序比原来的程序执行时间短。

```
int sum_array ( int a[N][N][N])
{
    int i, j, k, sum=0;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            for (k=0; k<N; k++) sum+=a[k][i][j];
    return sum;
}
```

13. 分析比较以下三个函数中数组访问的空间局部性，并指出哪个最好，哪个最差？

```
#define N 1000
typedef struct {
      int vel[3];
      int acc[3];
  } point;
point p[N];
void clear1(point * p, int n)
{
  int i, j;
  for (i=0; i<n; i++) {
      for (j=0; j<3; j++)
           p[i].vel[j]=0;
      for (j=0; i<3; j++)
           p[i].acc[j]=0;
  }
}
```

```
#define N 1000
typedef struct {
      int vel[3];
      int acc[3];
  } point;
point p[N];
void clear2(point * p, int n)
{
  int i, j;
    for (i=0; i<n; i++) {
      for (j=0; j<3; j++) {
           p[i].vel[j]=0;
           p[i].acc[j]=0;
      }
    }
}
```

```
#define N 1000
typedef struct {
      int vel[3];
      int acc[3];
  } point;
point p[N];
void clear3(point * p, int n)
{
  int i, j;
    for (j=0; j<3; j++) {
      for (i=0; i<n; i++)
           p[i].vel[j]=0;
      for (i=0; i<n; i++)
           p[i].acc[j]=0;
    }
}
```

14. 以下是计算两个向量点积的程序段。

```
float dotproduct (float x[8], float y[8])
{
    float sum=0.0;
    int i;
    for (i=0; i<8; i++) sum+=x[i] * y[i];
    return sum;
}
```

要求：

(1) 试分析该段代码中访问数组 x 和 y 的时间局部性和空间局部性，并推断命中率的高低。

(2) 假定该段程序运行的计算机中数据 cache 采用直接映射方式，其数据区容量为 32 字节，每个主存块大小为 16 字节。假定编译程序将变量 sum 和 i 分配给寄存器，数组 x 存放在 00000040H 开始的 32 字节的连续存储区中，数组 y 紧跟在 x 后进行存放。试计算该程序中数据访问的命中率，要求说明每次访问时 cache 的命中情况。

(3) 将上述(2)中的数据 cache 改用 2-路组相联映射方式，块大小改为 8 字节，其他条件不变，则该程序数据访问的命中率是多少？

(4) 上述(2)中条件不变的情况下，如果将数组 x 定义为 float[12]，则数据访问的命中率又是多少？

15. 以下是对矩阵进行转置的程序段。

```
typedef int array[4][4];
void transpose(array dst, array src)
{
    int i, j;
    for (i=0; i<4; i++)
        for (j=0; j<4; j++)
            dst[j][i]=src[i][j];
}
```

假设该段程序运行的计算机中 sizeof(int)=4,且只有一级 cache,其中 L1 data cache 的数据区大小为 32B,采用直接映射、回写方式,块大小为 16B,初始为空。数组 dst 从地址 0000C000H 开始存放,数组 src 从地址 0000C040H 开始存放。填写表 4.2,说明对数组元素 src[row][col]和 dst[row][col]的访问是命中(Hit)还是缺失(Miss)? 若 L1 data cache 的数据区容量改为 128B 时,重新填写表 4.2 的内容。

表 4.2 题 15 用表

	src 数组				dst 数组			
	col=0	col=1	col=2	col=3	col=0	col=1	col=2	col=3
row=0	Miss				Miss			
row=1								
row=2								
row=3								

16. 通过对方格中每个点设置相应的 CMYK 值就可以将方格图上相应的颜色。以下三个程序段都可实现对一个 8×8 的方格中图上黄色的功能。

```
struct pt_color {
            int c;
            int m;
            int y;
            int k;
}
struct pt_color square[8][8];
int i, j;
for (i=0; i<8; i++) {
    for (j=0; j<8; j++) {
        square[i][j].c=0;
        square[i][j].m=0;
        square[i][j].y=1;
        square[i][j].k=0;
    }
}
```

程序段 A

```
struct pt_color {
            int c;
            int m;
            int y;
            int k;
}
struct pt_color quare[8][8];
int i, j;
for (i=0; i<8; i++) {
    for (j=0; j<8; j++) {
        square [j][i].c=0;
        square [j][i].m=0;
        square [j][i].y=1;
        square [j][i].k=0;
    }
}
```

程序段 B

```
struct pt_color {
            int c;
            int m;
            int y;
            int k;
}
struct pt_color square[8][8];
int i, j;
for (i=0; i<8; i++)
    for (j=0; j<8; j++)
        square[i][j].y=1;
for (i=0; i<8; i++)
    for (j=0; j<8; j++) {
        square[i][j].c=0;
        square[i][j].m=0;
        square[i][j].k=0;
    }
```

程序段 C

假设 cache 的数据区大小为 512B,采用直接映射,块大小为 32B,存储器按字节编址,sizeof(int)=4。

编译时变量 i 和 j 分配在寄存器中，数组 square 按行优先方式存放在 000008C0H 开始的连续区域中，主存地址为 32 位。要求：

(1) 对三个程序段 A、B、C 中数组访问的时间局部性和空间局部性进行分析比较。

(2) 画出主存中的数组元素和 cache 中行的对应关系图。

(3) 分别计算三个程序段 A、B、C 中的写操作次数、写不命中次数和写缺失率。

17. 假设某计算机的主存地址空间大小为 64MB，采用字节编址方式。其 cache 数据区容量为 4KB，采用 4 路组相联映射方式、LRU 替换算法和回写(write back)策略，块大小为 64B。请回答以下问题。

(1) 主存地址字段如何划分？要求说明每个字段的含义、位数和在主存地址中的位置。

(2) 该 cache 的总容量有多少位？

(3) 假设 cache 初始为空，CPU 依次从 0 号地址单元顺序访问到 4344 号单元，重复按此序列共访问 16 次。若 cache 命中时间为一个时钟周期，缺失损失为 10 个时钟周期，则 CPU 访存的平均时间为多少时钟周期？

18. 假定某处理器可通过软件对高速缓存设置不同的写策略，那么，在下列两种情况下，应分别设置成什么写策略？为什么？

(1) 处理器主要运行包含大量存储器写操作的数据访问密集型应用。

(2) 处理器运行程序的性质与(1)相同，但安全性要求很高，不允许有任何数据不一致的情况发生。

19. 已知 cache 1 采用直接映射方式，共 16 行，块大小为一个字，缺失损失为 8 个时钟周期；cache 2 也采用直接映射方式，共 4 行，块大小为 4 个字，缺失损失为 11 个时钟周期。假定开始时 cache 为空，采用字编址方式。要求找出一个访问地址序列，使得 cache 2 具有更低的缺失率，但总的缺失损失反而比 cache 1 大。

20. 提高关联度通常会降低缺失率，但并不总是这样。请给出一个地址访问序列，使得采用 LRU 替换算法的 2-路组相联映射 cache 比具有同样大小的直接映射 cache 的缺失率更高。

21. 假定有三个处理器，分别带有以下不同的 cache。

cache 1：采用直接映射方式，块大小为一个字，指令和数据的缺失率分别为 4%和 6%。

cache 2：采用直接映射方式，块大小为 4 个字，指令和数据的缺失率分别为 2%和 4%。

cache 3：采用 2-路组相联映射方式，块大小为 4 个字，指令和数据的缺失率分别为 2%和 3%。

在这些处理器上运行同一个程序，其中有一半是访存指令，在三个处理器上测得该程序的 CPI 都为 2.0。已知处理器 1 和 2 的时钟周期都为 420ps，处理器 3 的时钟周期为 450ps。若缺失损失为(块大小+6)个时钟周期，请问：哪个处理器因 cache 缺失而引起的额外开销最大？哪个处理器执行速度最快？

22. 假定某处理器带有一个数据区容量为 256B 的 cache，其块大小为 32B。以下 C 语言程序段运行在该处理器上，设 sizeof(int)=4，编译器将变量 i，j，c，s 都分配在通用寄存器中，因此，只要考虑数组元素的访存情况。若 cache 采用直接映射方式，则当 s=64 和 s=63 时，缺失率分别为多少？若 cache 采用 2-路组相联映射方式，则当 s=64 和 s=63 时，缺失率又分别为多少？

```
int i, j, c, s, a[128];
⋮
for(i=0; i<10000; i++)
    for(j=0; j<128; j=j+s)
        c=a[j];
```

23. 假定一个虚拟存储系统的虚拟地址为 40 位，物理地址为 36 位，页大小为 16KB，按字节编址。若页表中有有效位、存储保护位、修改位、使用位，共占 4 位，磁盘地址不记录在页表中，则该存储系统中每个进程的页表大小为多少？如果按计算出来的实际大小构建页表，则会出现什么问题？

24. 假定一个计算机系统中有一个 TLB 和一个 L1 Data Cache。该系统按字节编址，虚拟地址 16 位，

物理地址12位，页大小为128B；TLB采用4路组相联方式，共有16个页表项；L1 Data Cache采用直接映射方式，块大小为4B，共16行。在系统运行到某一时刻时，TLB、页表和L1 Data Cache中的部分内容如图4.47所示。

组号	标记	页框号	有效位	标记	页框号	有效位	标记	页框号	有效位	标记	页框号	有效位
0	03	–	0	09	1D	1	00	–	0	07	10	1
1	13	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	63	12	1	0A	34	1	72	–	0

(a) TLB (4路组相联)：4组、16个页表项

虚页号	页框号	有效位
000	08	1
001	03	1
002	14	1
003	02	1
004	–	0
005	16	1
006	–	0
007	07	1
008	13	1
009	17	1
00A	09	1
00B	–	0
00C	19	1
00D	–	0
00E	11	1
00F	0D	1

(b) 部分页表：(开始16项)

行索引	标记	有效位	字节3	字节2	字节1	字节0
0	19	1	12	56	C9	AC
1	15	0	–	–	–	–
2	1B	1	03	45	12	CD
3	36	0	–	–	–	–
4	32	1	23	34	C2	2A
5	0D	1	46	67	23	3D
6	–	0	–	–	–	–
7	10	1	12	54	65	DC
8	24	1	23	62	12	3A
9	2D	0	–	–	–	–
A	2D	1	43	62	23	C3
B	–	0	–	–	–	–
C	12	1	76	83	21	35
D	16	1	A3	F4	23	11
E	33	1	2D	4A	45	55
F	14	0	–	–	–	–

(c) L1 Data Cache：直接映射，共16行，块大小为4B

图4.47 题24用图

请回答以下问题。

(1) 虚拟地址中哪几位表示虚拟页号？哪几位表示页内偏移量？虚拟页号中哪几位表示TLB标记？哪几位表示TLB索引？

(2) 物理地址中哪几位表示物理页号？哪几位表示页内偏移量？

(3) 主存物理地址如何划分成标记字段、行索引字段和块内地址字段？

(4) CPU从地址067AH中取出的值为多少？说明CPU读取地址067AH中内容的过程。

25. 缓冲区溢出是指分配的某个内存区域(缓冲区)的大小比存放内容所需空间小。例如，在栈中分配了一块空间用于存放某个过程的一个字符串，结果字符串长度超过了分配空间的大小。黑客往往会利用缓冲区溢出来植入入侵代码。请说明可以采用什么措施来防止缓冲区溢出漏洞。

第5章 指令系统

第1章提到，计算机硬件只能识别和理解机器语言程序，用各种高级语言编写的源程序最后都要翻译（汇编、解释或编译）成以指令形式表示的机器语言才能在计算机上执行。计算机的指令有微指令、机器指令和伪（宏）指令之分。微指令是微程序级命令，属于硬件范畴；伪指令是由若干机器指令组成的指令序列，属于软件范畴；机器指令介于二者之间，处于硬件和软件的交界面。本章中的“指令”如无特殊说明，都指机器指令。

一台计算机能执行的机器指令的集合称为该机的指令集或指令系统，它是构成程序的基本元素，也是硬件设计的依据，它衡量机器硬件的功能，反映硬件对软件支持的程度。系统软件直接建立在硬件支持的指令基础上，系统程序员感觉到的计算机的功能特性和概念性结构就是指令集体系结构（ISA），因此，ISA设计的好坏直接决定计算机的性能和成本，而指令系统是ISA中最核心的部分，因而指令系统的设计至关重要。

本章介绍指令系统设计中涉及的各个方面，主要包括指令格式、操作类型、操作数类型、寻址方式、操作码编码、指令系统的风格以及程序的机器级表示等。

5.1 指令格式设计

5.1.1 指令地址码的个数

冯·诺依曼结构计算机采用“存储程序”的工作方式。根据“存储程序”思想，计算机中的程序一旦被启动运行，则必须能够自动地逐条从主存取出指令执行。也就是说，不仅每条指令的执行过程是自动的，而且在一条指令执行结束后能够自动转到下一条指令执行。为此，一条指令中必须明显或隐含地包含以下信息。

(1) 操作码。指定操作类型，如移位、加减乘除、传送等。

(2) 源操作数或其地址。指出一个或多个源操作数或其所在的地址，可以是主（虚）存地址、寄存器编号或I/O端口，也可在指令中直接给出一个立即数。

(3) 结果的地址。产生结果所存放的地址，可以是主（虚）存地址、寄存器编号或I/O端口。

(4) 下条指令地址。下条指令存放的地址，可以是主（虚）存地址。

有了上述第(1)到(3)项信息，就可以完成一条指令的自动执行，有了第(4)项信息，就可以周而复始地自动执行一条条指令。

通常，下条指令的地址不需要在指令中明显给出，而是隐含在PC中。指令按顺序执行时，只要自动将PC的值加上指令的长度，就可以得到下条指令的地址，当遇到转移指令而不按顺序执行时，需由指令给出转移到的目标地址。

综上所述可知，一条指令由一个操作码和几个地址码构成。根据包含的地址个数可分为以下几类。

（1）三地址指令

分别作为双目运算中两个源操作数的地址和一个结果的地址。格式如下：

OP	D1	D2	D3

（D1） OP （D2） →D3

（2）二地址指令

分别存放双目运算中两个操作数，并将其中一个地址作为结果的地址。格式如下：

OP	D1	D2

（D1） OP （D2） →D2

（3）单地址指令

对于单目运算（如取反/取负等），其地址既是操作数的地址，也是结果的地址。对于双目运算，则另一个操作数和结果默认存放在累加器A中，格式如下：

OP	D

（D） OP（A） →A

（4）零地址指令

这类指令无操作数，所以无地址码，例如，空操作、停机等不需要地址的指令；或者操作数隐含在堆栈中，其地址由栈指针给出的指令等都是零地址指令。

5.1.2 指令格式设计原则

指令格式的选择应遵循如下几条基本原则。

（1）指令应尽量短。指令长度短，使得程序占用存储空间小，降低空间开销。

（2）要有足够的操作码位数。向后兼容使指令操作类型不断增加，因此必须预留足够的操作码位数。

（3）操作码的编码必须有唯一的解释。操作码最终需送到指令译码器进行译码，因此，指令操作码要么是一个唯一的合法编码，要么是不合法的序列。当译码器发现是不合法操作码时，出现“非法指令”异常。

（4）指令长度应是字节的整数倍。指令存放在内存，而内存往往按字节编址，因此指令长度为字节的整数倍，便于指令的读取和指令地址的计算。

（5）合理选择地址字段的个数。地址字段个数涉及到指令的长度和指令的规整性问题，它是空间开销和时间开销权衡的结果。

（6）指令应尽量规整。指令的规整性体现在许多方面：指令长度是否固定、操作码位数是否固定、地址码格式是否一致、指令字中各字段的划分位置是否一致等。规整的指令系统会大大简化硬件的实现。

5.2 指令系统设计

指令系统设计是计算机系统结构设计的关键之一。在设计指令系统时，必须遵循以下基本原则。

(1) 完备性或完整性。指令的操作类型应尽量完备，应能足够编制任何可计算程序。但是，如果指令系统太复杂，也会给硬件实现增加困难。因此，较复杂的功能可以通过伪指令实现。

(2) 兼容性。在考虑系列机设计实现时，高档机的指令系统应兼容以前低档机的指令系统，这给软件资源重复利用带来方便。

(3) 均匀性。运算指令应能对多种类型的数据进行处理，包括三种整数(字节、字、双字)和两种浮点数(单精度和双精度浮点数)类型。

(4) 可扩充性。操作码字段要预留一定的编码空间，以便需要时进行扩充。

5.2.1 基本设计问题

在设计一个指令系统时，需要考虑以下一些基本问题。

(1) 操作码的个数、种类、复杂度如何选择?

例如，若指令系统中共包含 4 条指令：取数指令(Load)、存数指令(Store)、自增指令(INC)、分支指令(BRN)，则用这 4 条指令足以编制任何可计算程序。虽然这 4 条指令是完备的，但会导致大多数程序变得很长，既占空间又费时间。

(2) 运算指令能对哪几种数据类型进行操作?

高级语言源程序中需要对 int、short 和 byte 等类型的整型数据以及 float、double 等浮点类型数据，甚至是位串、字符串等进行操作，所以，指令设计时也需要考虑能对这些数据类型进行操作。

(3) 采用什么样的指令格式?

规整型指令采用定长指令字和定长操作码，使得取指令、指令译码、指令地址计算等变得简单，从而能减少时间开销，但是，规整型指令在空间上会增加开销。因此，应根据设计目标选择采用规整型还是紧凑型指令格式。

(4) 通用寄存器的个数、功能、长度等如何规定?

用户进程的指令中能用的寄存器是用户可见寄存器，也称为通用寄存器。通用寄存器多，则编译器可以尽量多地把高级语言源程序中的变量分配到通用寄存器中，因而，减少指令执行时访问内存的次数，加快程序运行。但是，通用寄存器多会使寄存器存取延迟变长，因而影响指令执行速度。此外，还可能使寄存器编码变长，从而使指令长度变长，通用寄存器多还会增大 CPU 成本，占用更多硅片面积。

通用寄存器的功能分配也很重要，例如，要考虑是否要有专门的栈指针寄存器、栈帧指针寄存器、过程调用的参数寄存器、过程调用的返回参数寄存器、过程调用的返回地址寄存器；是否要有记录指令执行状态的标志寄存器等。

此外，还要考虑每个寄存器的长度以及寄存器的设计如何满足多种不同长度的数据类型。例如，Intel 体系结构 IA-32 中，用寄存器扩展的方式提供了存放 8 位、16 位、32 位等多

种长度操作数的寄存器；而MIPS等体系结构中采用的是固定寄存器宽度的设计方案，通过提供不同的指令来区分操作数的长度。

(5) 如何设计寻址方式的种类、寻址方式字段的编码以及各种寻址方式下有效地址如何计算？

寻址方式字段可以和操作码一起编码，由操作码确定每个操作数的寻址方式，如，MIPS体系结构就由操作码确定指令类型是R型、I型还是J型，指令类型确定后，每个操作数的寻址方式就确定了；寻址方式字段也可单独编码，例如，IA-32指令中每个操作数都有各自专门的寻址方式字段。

(6) 下条指令的地址如何确定？

几乎所有指令系统都通过一个专门的寄存器来存放下条指令的地址，这个寄存器被称为程序计数器(Program Counter，PC)或指令指针(Instruction Pointer，IP)。

顺序执行时，指令中无须明显地给出下条指令地址，默认由PC指出。通过将当前PC的值加上本条指令的长度即可得到下条指令地址；转移指令等可能改变程序执行顺序，此时，这些指令中必须有相应的地址码和寻址方式来给出下条指令地址或下条指令地址的计算方式。

当然，指令系统的设计所涉及的远远不限于上述所列问题，在具体设计过程中，还需要考虑很多细节问题。

5.2.2 操作数类型

操作数是指令处理的对象，根据高级语言程序所用的类型来看，指令涉及到的基本操作数类型应该包括以下几类。

(1) 地址。无符号整数，用来表示主(虚)存地址。

(2) 数值数据。带符号整数和浮点数。带符号整数一般用二进制补码表示，浮点数大多用IEEE 754标准表示。有些指令系统也提供十进制数运算指令，一般用NBCD码(8421码)表示十进制数。

(3) 位、位串、字符和字符串。位和位串数据一般用来表示一些标志、控制和状态等信息。字符和字符串数据用来表示文本、流式文件基本信息等。

(4) 逻辑(布尔)数据。表示逻辑值(0—假/1—真)。

例如，Pentium处理器提供的基本类型有字节、字(16位)、双字(32位)、四字(64位)。对于整数，有16位、32位、64位三种补码表示的整数和18位压缩BCD码表示的十进制整数；对于序数(即地址、指针等)，有字节、字或双字长的无符号整数；对于浮点数，有用IEEE 754表示的32位单精度浮点数、64位双精度浮点数和80位扩展精度浮点数。另外，还提供了专门的近指针类型数据，用于表示不分段存储器的地址，或用来表示段内偏移的32位有效地址，以进行分段存储器的段内访问。

有关以上各类数据的表示、存放和运算已在第2章和第3章中详细介绍。

5.2.3 寻址方式

指令不仅要规定所执行的操作，还要给出操作数或操作数地址。操作数可能是一个常数，或一个简单变量，或是数组和结构中的某个元素，也可能是栈(stack)中的元素，还

可能是外设I/O接口中的状态字或控制字等。从指令的角度来看，操作数存放位置可以是CPU中的通用寄存器、存储单元和I/O端口。通常把指令中给出的操作数所在存储单元的地址称为有效地址，存储单元地址可能是主存物理地址，也可能是虚拟地址。如果不采用虚拟存储机制，有效地址就是主存物理地址；若采用虚拟存储机制，则有效地址就是虚拟地址。

指令给出操作数或操作数地址的方式称为寻址方式。地址字段长度直接影响指令长度，因而指令地址码要尽量短，但操作数的存放位置又必须灵活，存放空间也应尽量大。因此，指令系统应能提供灵活的寻址方式，并使用尽量短的地址码访问尽可能大的寻址空间。此外，为加快指令执行速度，有效地址计算过程也应尽量简单。

常用的寻址方式有以下几种。

1. 立即寻址

指令中直接给出操作数本身，这种操作数称为立即数。

2. 直接寻址

指令中直接给出操作数的有效地址，这种地址称为直接地址或绝对地址。

格式如下：

OP	操作数地址

3. 间接寻址

指令中给出存放操作数有效地址的地址。如图5.1所示的是单级间接寻址过程，还可有多重间址。格式中的@是间接寻址标志。

格式如下：

OP	@间接地址

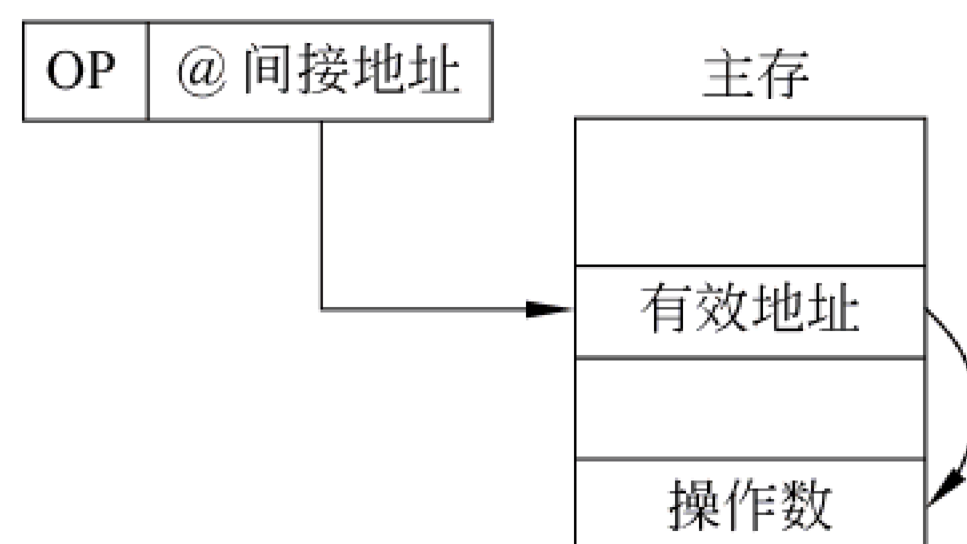


图5.1 单级间接寻址

4. 寄存器寻址

指令中给出操作数所在的寄存器编号，操作数在寄存器中。

格式如下：

OP	寄存器号

或

OP	寄存器号1	寄存器号2

上面给出的两种格式中，前一种是单操作数指令或者两个操作数中有一个隐含给出(如累加器)；后一种为两个操作数均在寄存器中。

寄存器寻址有以下优点。

(1) 寄存器数量远小于内存单元数，所以寄存器编号比存储器地址短，因而寄存器寻址方式下的指令较短。

(2) 操作数已在CPU中，不用访存，因而指令执行速度快。

5. 寄存器间接寻址

操作数的有效地址在指令指定的某个寄存器中。如 8086 指令“MOV AX,[BX]”中,寄存器 BX 内容为有效地址,该有效地址中的内容是操作数,被送入 AX 寄存器中。寄存器间接寻址指令也短,因为只要给出一个寄存器编号而不必给出有效地址。指令长度和寄存器寻址指令差不多,但由于要访存,所以寄存器间接寻址指令的执行时间比寄存器寻址指令的执行时间更长。

6. 变址寻址

变址寻址方式主要用于对线性表之类的数组元素进行方便的访问。采用变址寻址方式时,指令中的地址码字段 A 给出一个基准地址,例如,数组的起始地址,而数组元素相对于基准地址的偏移量在指令中明显或隐含地由变址寄存器 I 给出,这样,变址寄存器的内容实际上就相当于数组元素的下标,每个数据元素的有效地址为基准地址加变址寄存器的内容,即操作数的有效地址 EA=(I)+A,其中(I)表示变址寄存器 I 的内容。

如果任何一个通用寄存器都可作为变址寄存器,则必须在指令中明确地给出一个通用寄存器的编号,并标明作为变址寄存器用;若处理器中有一个专门的变址寄存器,则无须在指令中明确给出。指令中的地址码字段称为形式地址,因而这里的形式地址是基准地址 A,而变址寄存器中的偏移量应该是一个无符号数。

例如,8086 指令“MOV AL,[SI+1000H]”中的 SI 为变址器,1000 为形式地址,SI 的内容加上 1000H 形成操作数的有效地址。

如图 5.2 所示,指令中的地址码 A 给定数组在存储器中的首地址,变址器 I 每次自动加(减)数组元素的长度。当数组元素的访问沿低地址向高地址方向时,自动加;当访问沿高地址向低地址方向时,自动减。若存储器按字节编址,每个数组元素为一个字节时,则 I=(I) ± 1;若每个元素为 4 个字节时,则 I=(I) ± 4。这样,在程序的循环体中,第一次变址寄存器 I 的值初始化为 0,执行取数指令取出 A[0]后,I 自动变为 1,下次执行循环体时,取数指令就能取出 A[1],…,如此循环,直到取出所需的每个数组元素。

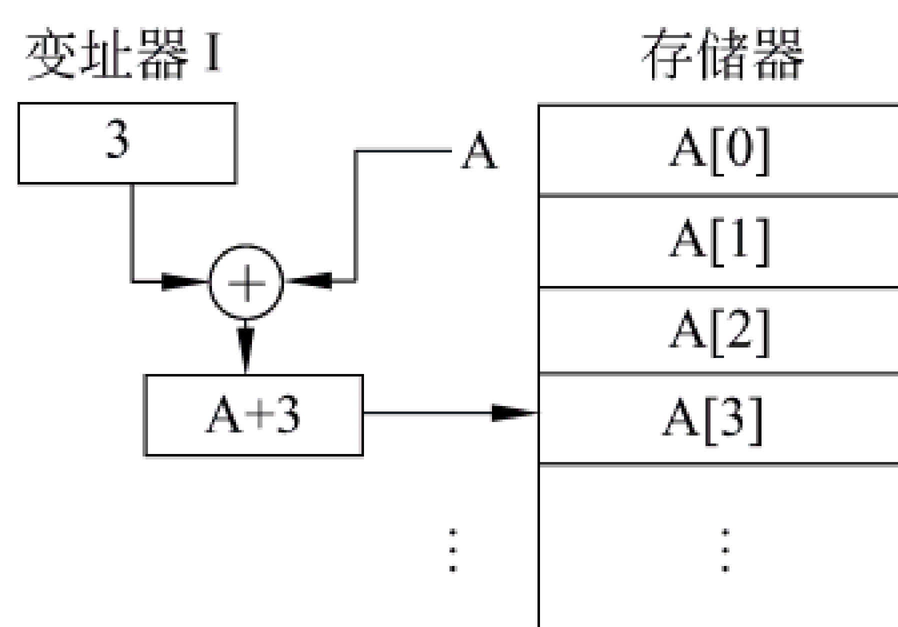


图 5.2 数组元素的变址寻址

通过自动变址,可以在没有硬堆栈的情况下,用它来建立软堆栈。

某些计算机中还允许变址与间址结合使用。假定指令中给出的变址寄存器为 I,形式地址为 A,则先变址后间址时,操作数的有效地址为 EA=((I)+A),称为前变址;先间址后变址时,则 EA =(I)+(A),称为后变址。

7. 相对寻址

如果某指令的操作数的有效地址或转移目标地址位于该指令所在位置的前、后某个固定位置上,则该操作数或转移目标可用相对寻址方式。采用相对寻址方式时,指令中的地址码字段 A 给出一个偏移量,基准地址隐含由 PC 给出。也即操作数有效地址或转移目标地址 EA=(PC)+A。这里的偏移量 A 是形式地址,有效地址或目标地址可以在当前指令之前或之后,因而偏移量 A 是一个带符号整数。

显然,相对寻址方式可用来实现公共子程序的浮动或实现相对转移。

8. 基址寻址

基址寻址方式下，指令中的地址码字段 A 给出一个偏移量，基准地址可以明显或隐含地由基址寄存器 B 给出。操作数有效地址 EA＝(B)＋A。与变址方式一样，若任意一个通用寄存器都可用作基址寄存器，则指令中必须明确给出通用寄存器编号，并标明用作基址寄存器。

基址寻址看起来与变址寻址相同，都是把某寄存器的内容和指令给出的形式地址相加得到有效地址，但实际功能和用法却不同。变址寻址主要解决数组元素的循环访问问题，指令中给出的形式地址是数组的首地址；而基址寻址可用于程序的重定位。前面提到过，在多道程序系统中，每个用户进程都在各自独立的虚拟地址空间中编写程序，但所有进程在系统中都是装入到一个统一的主存空间中运行的，因而需要在主存空间中对每个进程重新定位。通常是给定一个基地址，进程中的代码和数据被装入到该基地址开始的一块主存区域。对该进程中的代码和数据的访问都可以用指令中的形式地址和基地址相加来实现。在基址寻址方式中，基址寄存器的内容要求能对整个地址空间进行寻址，指令中给出的形式地址实际上指出了相对于基地址的偏移量。

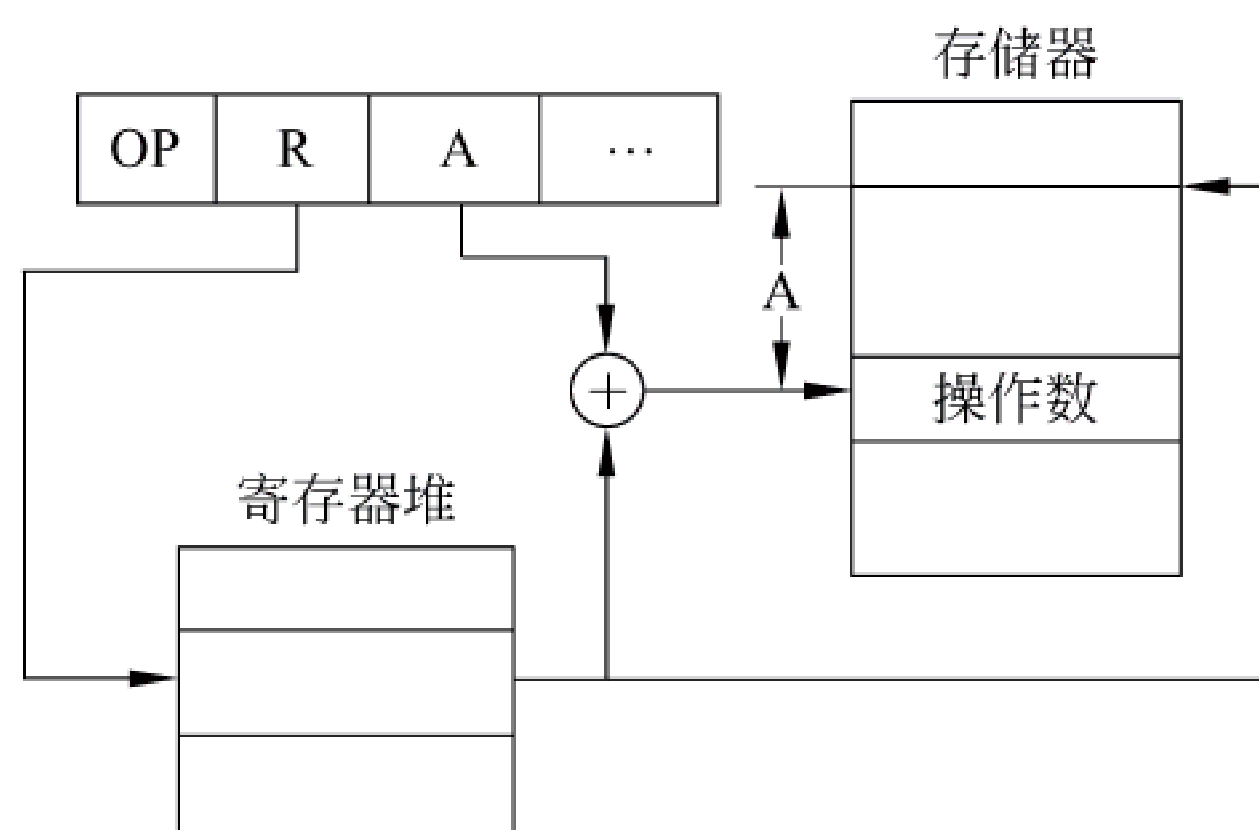


图 5.3　基址寻址过程

基址寻址过程如图 5.3 所示，其中，基址寄存器 R 可以指定为任何一个通用寄存器。寄存器 R 的内容是基准地址，加上形式地址 A，形成操作数有效地址。基址寻址为逻辑地址到物理地址变换提供了支持，用以实现程序的动态重定位。

变址、基址和相对三种寻址方式非常类似，都是将某个寄存器的内容与一个形式地址相加来生成操作数的有效地址。通常把它们统称为偏移寻址。

9. 其他寻址方式

为缩短指令字长度，有些指令采用隐含地址码方式。即在指令中不明显给出操作数地址或变址寄存器和基址寄存器编号，而是由操作码隐含指出。例如，单地址指令中只给出一个操作数地址，另一个操作数隐含规定为累加器内容；此外，还有堆栈操作指令，其操作数隐含为栈顶元素，指令中无须明显指出操作数地址。

有些计算机的指令系统还有更复杂的寻址方式，如位寻址、块寻址、串寻址等。

位寻址：当需要对寄存器或主(虚)存中单独一位进行操作(如置位/复位/测试等)时，需要进行位寻址。指令中必须隐含或明显地给出位指针，以指定对哪一位进行操作。

字节寻址：当操作数为一个字节时，指令必须对字节进行定位。字节编址方式时，指令中必须指出访问的是字节、半字、字还是双字；字编址方式时，指令必须指出是否为字节访问，并指出是哪个字节。现代计算机一般采用字节编址方式。

块寻址：当需对一个信息块进行操作时，指令必须对块进行定位。如 VAX 11/780 指令系统中就有块操作指令。此时，指令可采用“首址＋块长度”、“首址＋末址”或“首址＋末端标志”的方式来指定一个块。

5.2.4 操作类型

指令系统的完备性要求在设计指令系统时必须考虑指令系统应提供哪些操作类型。对大多数指令系统考察后得知，指令操作类型按功能分为以下几种。

1. 算术和逻辑运算指令

这类指令有加(ADD)、减(SUB)、比较(CMP)、乘(MUL)、除(DIV)、与(AND)、或(OR)、取反(NOT)、变补(NEG)、异或(XOR)、加1(INC)、减1(DEC)等。为了方便多字长数据的运算，大多数机器还设置了带进位的加(ADC)和带借位的减(SBB)指令等。在算术运算指令中，有的计算机还专门设置了十进制数的运算指令。

2. 移位指令

这类指令有算术移位、逻辑移位、循环移位、半字交换等。有的机器默认一条指令只移一位，如果要移多位的话，需要多条移位指令；有的机器可在指令中规定移动的位数，这样的话，移动多位的功能可以用一条指令实现。通常用一个桶形移位器实现一次移动多位的功能。

各种移位操作的含义如下。

算术左移：操作数的各位依次向左移，低位补零。有些机器将原操作数的最高位移入进位标志(CF)位，这样，通过判断符号标志和进位标志是否相等就可判断是否发生了溢出。

算术右移：各位依次向右移，高位补符号。有些机器将最低位移入进位标志位。

逻辑左移：操作同算术左移，大多数机器一般不再专门设置此指令。

逻辑右移：各位依次向右移，高位补零。有些机器将原操作数最低位移入进位标志位。

小循环左移：最高位移入进位标志位，同时也移入最低位。

小循环右移：最低位移入进位标志位，同时也移入最高位。

大循环左移：最高位移入进位标志位，而进位标志位移入最低位。

大循环右移：最低位移入进位标志位，而进位标志位移入最高位。

半字交换：寄存器的前半部分和后半部分内容进行交换。

3. 传送指令

有寄存器间传送(MOV)、取数(LOAD：内存→CPU 寄存器)、存数(STORE：CPU 寄存器→内存)等。

4. 串指令

对字符串进行操作的指令。如串传送、比较、检索、传送转换等指令。

5. 顺序控制指令

用来控制程序执行的顺序。有条件转移(BRANCH)、无条件转移(JMP)、跳步(SKIP)、调用(CALL)、返回(RET)等指令。

顺序控制类指令的功能通过将转移目标地址送到 PC 中来实现。转移目标地址可用直接寻址方式给出(又称绝对转移)或由相对寻址方式给出(又称相对转移)。有的机器还可以用寄存器寻址方式或寄存器间接寻址方式给出转移目标地址。

无条件转移指令在任何情况下都执行转移操作，而条件转移指令(或称分支指令)仅仅在特定条件满足时才执行转移操作。转移条件一般是某个标志位的值，或者由两个或两个以上的标志位组合而成，例如，CF=1，CF=0 或 CF=1 且 ZF=1 等。这里 CF 为进位标

志,ZF 为零标志。跳步是转移的一种特例,它使 PC 再增加一个定值,这个定值一般是指令字所占用的存储字个数。应该注意的是取指令时 PC 已增量过了,因此跳步指令实际上就是跳过下条指令。

调用指令也称为转子指令,和转移指令的根本区别在于执行调用指令时必须保存下条指令的地址(称为返回地址)。调用指令用于子程序调用(即过程调用),当子程序执行结束时,根据返回地址返回到主程序继续执行;而转移指令则不返回执行,因而无须保存返回地址。

返回指令的功能是在子程序执行完毕时,将事先保存的返回地址送到 PC,这样处理器就能回到原来的主程序继续执行。

6. CPU 控制指令

这类指令有停机、开中断、关中断、系统模式切换以及进入特殊处理程序等指令。大多数机器将这类指令划为“特权”指令(也称为管态指令),只能用于内核进程中,用户进程一般不能使用,以防止因用户使用不当而对系统运行造成危害。

7. 输入、输出指令

这类指令用于完成 CPU 与外部设备交换数据或传送控制命令及状态信息。大多数机器都设置了这类指令,但是它们的寻址方式一般较少,常见的只有寄存器寻址、直接寻址和寄存器间接寻址等。当外设中的 I/O 地址空间和主存地址空间统一编址时,可以不设置这类指令,而用访存指令完成 I/O 操作。

5.2.5 操作码编码

指令的操作码字段可以是固定长度,也可以是可变长度。选择定长操作码还是可变长操作码,是时间和空间之间的开销权衡问题。希望降低空间开销时,代码的长度更重要,应采用紧凑的变长操作码和变长指令字;希望降低时间开销以取得更好性能时,应采用定长操作码和定长指令字。

1. 定长操作码编码

指令的操作码部分采用固定长度编码,这种方式译码方便,指令执行速度快,但有信息冗余。例如,IBM 360/370 采用 8 位定长操作码,最多可有 256 条指令,但指令系统中只提供了 183 条指令,有 73 种为冗余编码。如图 5.4 所示,IBM 360/370 的指令格式有 RR 型、RX 型、RS 型、SI 型、SS 型等几类指令。其中 RR 型指令为半字长(16 位),SS 型指令为一字半长(48 位),其余指令为单字长(32 位)。操作码的第 0 位和第 1 位组成 4 种不同的编码,代表 4 种不同的指令:00 表示 RR 型,01 表示 RX 型,10 表示 RS 型和 SI 型,11 表示 SS 型。RR型指令是寄存器-寄存器型指令,即两个操作数都是寄存器中的内容;RX 型指令和 RS 型指令都是寄存器-存储器型指令,其中 RX 型是二地址指令,第一个操作数和结果放在 R1 中,另一个操作数在存储器中,采用变址寻址方式,有效地址 EA=(X)+(B)+D;RS 型是三地址指令:R1 存放结果,R3 存放一个源操作数,另一个源操作数在存储器中,其有效地址 EA=(B)+D;SI 型是存储器-立即数型指令,其结果和其中一个操作数的地址共用同一个存储单元;SS 型指令是存储器-存储器型指令,即两个操作数都是存储器中的内容,用于字符串的运算和处理,L 为字符串的长度。

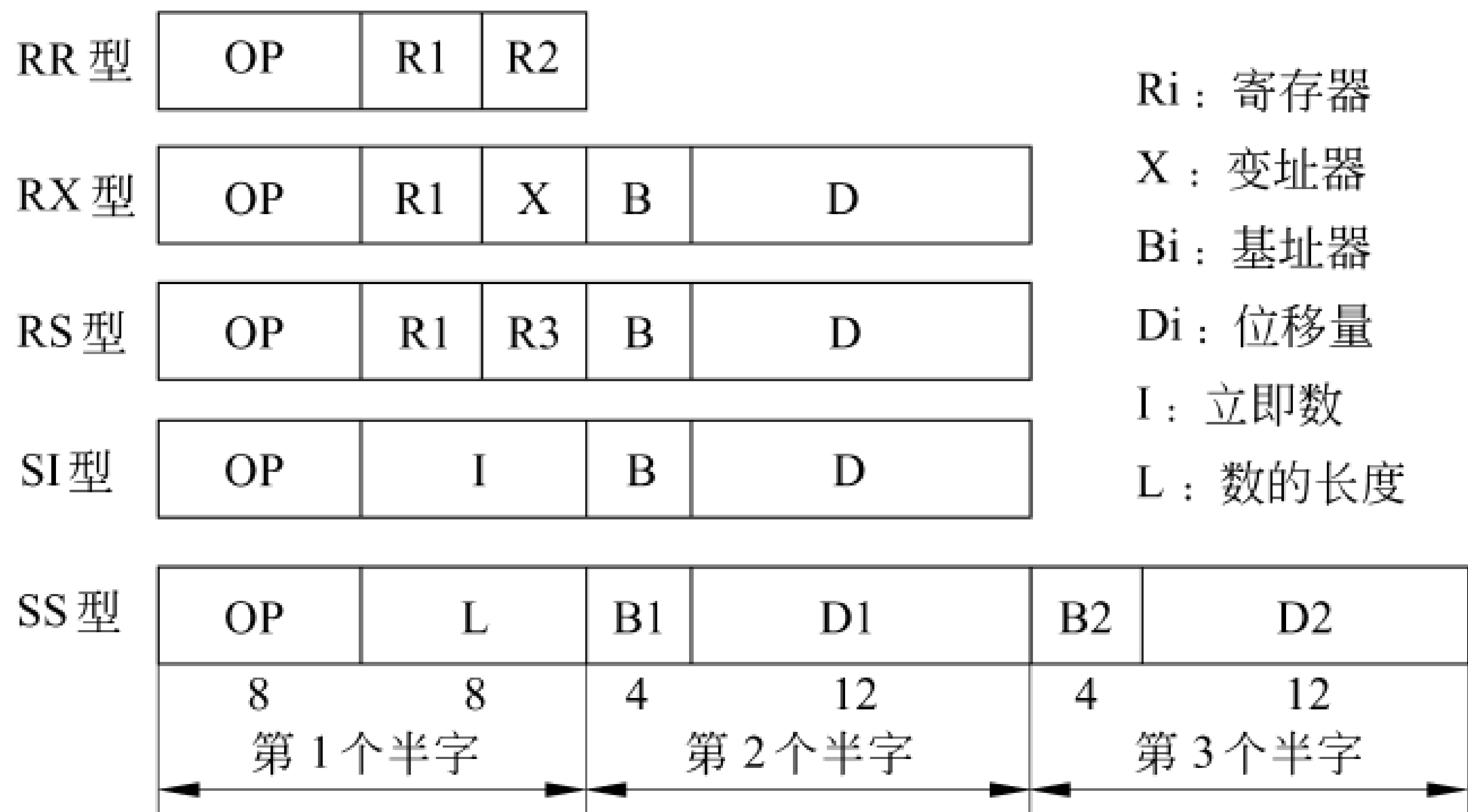


图5.4　IBM 360/370指令格式

2. 扩展操作码编码

扩展操作码编码方式将操作码的编码长度分成几种固定长度的格式。可以采用等长扩展法，例如，按4-8-12、3-6-9这种等步长方式扩展，也可采用不等长扩展法。扩展编码方式的操作码长度不固定，是可变的。这种编码方式被大多数非规整型指令集采用。

PDP-11是典型的变长操作码计算机，如图5.5所示，其各种操作码长度依次为4-7-8-10-13-16。

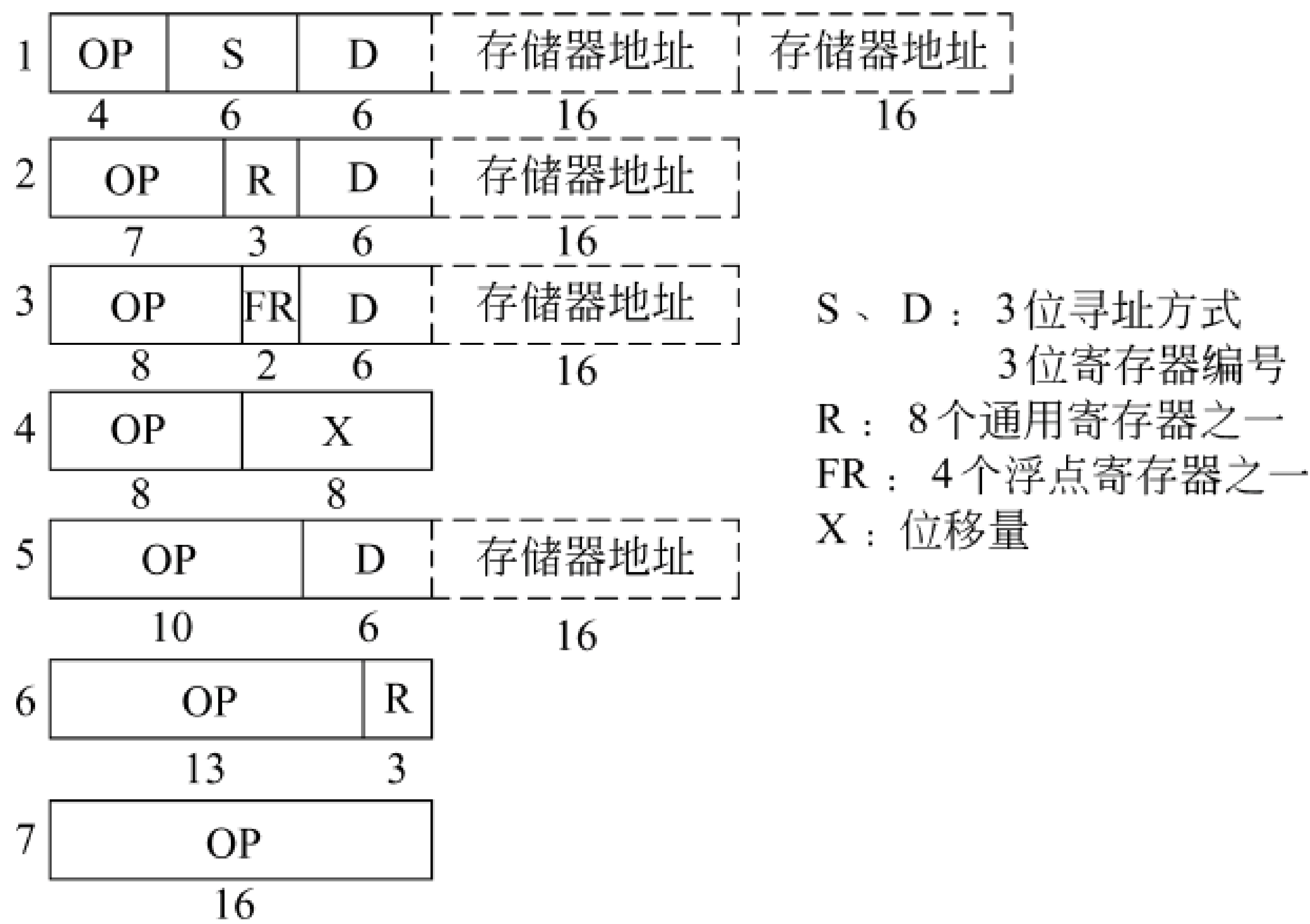


图5.5　PDP-11指令格式

PDP-11是16位字长的小型计算机，有8个寄存器R0～R7，其中，R0～R5为通用寄存器，R6是栈指针SP，R7是程序计数器PC。存储器按字节编址，要求边界对齐，每个字地址都是偶数。基本指令字长为16位，第1、2、3类为双操作数指令；第4类为转移指令，转移目标地址采用相对寻址方式；第5、6类为单操作数指令。有些类型的指令可扩充一个或两个存储器地址字段，扩充字段分别对应S和D字段，是否扩充取决于S和D中的寻址方式位。

下面用一个例子来说明如何进行扩展操作码编码。

例5.1　设某指令系统的指令字为16位，每个地址码为6位。若二地址指令15条，一地址指令34条，则剩下的零地址指令最多有多少条？

解：扩展编码的基本思想就是操作码按短到长进行扩展编码。二地址指令操作码最短，零地址指令的操作码最长，所以，按照二地址→一地址→零地址的顺序进行。

二地址指令的地址码部分占 12 位，故操作码只有 4 位，最多有 16 种编码，用去 15 种编码(0000～1110)分别表示 15 条指令，还剩一种编码 1111 未使用。

一地址指令的地址码部分占 6 位，故操作码有 10 位，最高 4 位为 1111，还剩 6 位，最多可有 $2^6=64$ 种编码，用其中的 $32+2=34$ 种编码(11110 00000～11110 11111 和 11111 00000～11111 00001)分别表示 34 条一地址指令。

剩下的零地址指令共有 16 位操作码，其中高 5 位只能是 11111，所以编码范围为：11111(00010～11111)(000000～111111)，因此，零地址指令最多有 30×2^6 种编码可用。

*5.2.6 条件码的生成与使用

条件转移指令(也称分支指令)通常根据程序当前生成的条件码(Condition Codes，CC)进行转移，条件码也称为状态位(Status)或标志位(Flags)。

常用的标志有 4 种。

(1) 符号标志 NF(Negative)(有些系统也用 SF(Sign)表示)，NF=1，表示结果为负数。

(2) 溢出标志 VF(Overflow)(有些系统也用 OF 表示)，VF=1，表示结果溢出。

(3) 进(借)位标志 CF(Carry)，CF=1，表示结果产生了进(借)位。

(4) 零标志 ZF(Zero)，ZF=1，表示结果为 0。

可通过执行算术指令或显式地由比较和测试指令来设置标志位，例如，以下两条指令可实现条件转移。

```
sub   r1, r2, r3;          r2 和 r3 相减，结果存在 r1 中，并生成各种标志位
bz    label;               判断是否 ZF=1，若是，则转移到 label 处执行
```

生成的标志位可由专门的条件码寄存器(或状态寄存器、标志寄存器、程序状态字寄存器[①])来存放，也可由指定的通用寄存器来存放。不同机器的说法和做法类似，但不一定完全相同。例如，8086/8088 处理器中通过一个专门的 16 位标志寄存器 flags 来记录标志信息以及人为设置的控制信息(也称为自陷允许标志)，80386 以后的标志寄存器又逐步扩展为 32 位，增加了一些新的标志。

有些处理器不用专门的标志寄存器存放标志位，而是用通用寄存器来保存。下面的例子中，用通用寄存器 r1 来存放标志位。

```
cmp   r1, r2, r3;          比较 r2 和 r3，标志位存储在 r1 中
bgt   r1, label;           根据 r1 中标志位判断是否大于，以转移到 label 处
```

有的指令系统可以用以下一条“计算并转移”指令来实现上述两条指令的功能。

```
bgt   r1, r2, label;         根据 r1 和 r2 比较的结果，直接决定是否转移
```

由此可见，处理条件码生成的方式有多种，实现条件转移的方式也可以各不相同。不管

① 程序状态字寄存器用来存放条件码 CC 和自陷允许标志(Trap Enable Flag)等状态信息。不同计算机对程序状态的描述以及程序状态存放位置可能不一样。但在概念上应该有一个程序状态字(Program Status Word，PSW)。

是否保存条件码，也不管将条件码保存在特殊的标志寄存器中还是通用寄存器中，处理器中的运算电路必须能够产生这些基本条件码。有关内容参见第3章的第3.3小节。

5.2.7 指令系统设计风格

1. 按操作数位置指定风格来分

按操作数位置指定风格来分，可分为以下4种不同风格类型的指令系统。

(1) 累加器(Accumulator)型指令系统

这种类型指令系统中，总是把其中一个操作数隐含在累加器(一般用AC表示)中，指令执行的结果也总是送到累加器中。这种指令系统的指令字短，但每次运算都要通过累加器，因而，在复杂表达式运算时，程序中会多出许多移入/移出累加器的指令，从而使程序变长，影响程序执行的效率。这种设计风格的指令系统只在早期机器中使用过，现在一般不采用。

(2) 堆栈(Stack)型指令系统

Java虚拟机采用的是堆栈型指令系统。堆栈是一种采用后进先出(LIFO)或先进后出(FILO)存取方式的特定的存储区。堆栈型指令系统中，规定指令的操作数总是来自堆栈的栈顶。往堆栈里存数叫入(进)栈或压栈，从堆栈里取数叫出栈或弹出。

可用移位寄存器实现堆栈，这是一种用硬件实现堆栈的方法，又叫栈顶固定方式堆栈；也可以在内存中开辟堆栈区，堆栈的底部固定，栈顶位置动态变化。在CPU中有专门的栈指针SP，用来指示栈顶位置，存取只能在栈顶进行。有从高地址向低地址增长的“自顶向下”和从低地址向高地址增长的“自底向上”两种形式的内存堆栈区。一般机器采用“自顶向下”形式的堆栈。

堆栈型指令系统中的指令都是零地址或一地址指令，因此，指令字很短。但是，由于指令所用操作数只能来自栈顶，所以，在对表达式进行编译时，所生成的指令顺序以及操作数在堆栈中的排列都有严格的顺序规定，因而不灵活，带来指令条数的增加。因此，堆栈型指令系统很少被通用计算机使用。

(3) 通用寄存器(General Purpose Register)型指令系统

这种类型指令系统的特点是，使用通用寄存器而不是累加器来存放运算过程中所用的临时数据。其指令的操作数可以是立即数(I)，或来自通用寄存器(R)，或来自存储单元(S)。指令类型可以是RR型、RS型、SI型、SS型等。这类指令系统最典型的代表是IA-32，此外，还有Motorola 68xxx、VAX 11/780等。

(4) Load/Store型指令系统

Load/Store型指令系统也使用通用寄存器而不是累加器来存放运算过程中所用的临时数据。因此，它也是一种通用寄存器型指令系统。同时，它有一个显著的特点，就是只有取数(Load)指令和存数(Store)指令才可以访问存储器，运算类指令不能访存，也就是说运算类指令只能是RR型或RI型。Load/Store型指令系统中的指令比较规整，体现在每条指令的指令字长度和指令执行时间等能够比较一致。

目前，通用寄存器型指令系统占主导地位。这主要因为：①通用寄存器和处理器集成在一起，作为ALU的操作数来源，两者可以靠得很近，因而，可缩短传输延迟；②寄存器位于存储器层次化结构的顶端，速度快且容易使用。寄存器个数不能太多，否则，成本高且会延长存取时间而使得时钟周期变长。当然，寄存器个数也不能太少，否则，编译器只能把许

多变量分配到内存单元，每次都要去内存访问操作数，因而会影响程序的性能。因此，通用寄存器的设计和有效使用是程序性能好坏的关键之一。

2. 按指令格式的复杂度来分

按指令格式的复杂度来分，可分为 CISC 与 RISC 两种类型指令系统。

(1) CISC 风格指令系统

随着 VLSI 技术的迅速发展，计算机硬件成本不断下降，软件成本不断上升。为此，人们在设计指令系统时增加了越来越多功能强大的复杂命令，以使机器指令的功能接近高级语言语句的功能，给软件提供较好的支持。例如，VAX 11/780 指令系统包含了 16 种寻址方式，9 种数据格式，303 条指令，而且一条指令包含 1～2 个字节的操作码和下续 N 个操作数说明符，而一个操作数说明符的长度可达 1～10 个字节。我们称这类计算机为复杂指令集计算机(Complex Instruction Set Computer，CISC)。

CISC 指令系统设计的主要特点如下。

① 指令系统复杂：指令多、寻址方式多、指令格式多。

② 指令周期长：绝大多数指令需要多个时钟周期才能完成。

③ 指令周期差距大：各种指令都能访问存储器，使得简单指令和复杂指令所用的时钟周期数相差很大，不利于指令流水线的实现。

④ 采用微程序控制：由于有些指令非常复杂，以至于无法用组合逻辑控制器来实现，而微程序控制器用软件设计思想实现硬件，可以实现对复杂指令的控制。

⑤ 难以进行编译优化：由于编译器可选指令序列增多，使得目标代码组合增加，从而增加了目标代码优化的难度。

复杂的指令系统使得计算机的结构也越来越复杂，不仅增加了研制周期和成本，而且难以保证其正确性，甚至降低了系统性能。

对大量典型的 CISC 程序调查结果表明，各种指令的使用频率相当悬殊，最常使用的是只占指令系统 20%的一些简单指令，它们占程序代码的 80%以上，而需要大量硬件支持的复杂指令在程序中的出现频率却很低，造成了硬件资源的大量浪费。在微程序控制的计算机中，占程序中指令总数 20%的最复杂的指令占用了微程序控制存储器容量的 80%。因此，1975 年IBM 公司开始研究指令系统的合理性问题，John Cocke 领导的一个研究小组提出了精简指令集计算机(Reduced Instruction Set Computer，RISC)的概念。

(2) RISC 风格指令系统

RISC 的着眼点不是简单地放在简化指令系统上，而是通过简化指令使计算机结构更加简单合理，从而提高机器的性能。与 CISC 相比，RISC 指令系统的主要特点如下。

① 指令数目少：只包含频度高的简单指令。

② 指令格式规整：寻址方式少、指令格式少、指令长度一致。

③ 采用 Load/Store 型指令设计风格。

④ 采用流水线方式执行指令：规整的指令格式有利于采用流水线方式执行，除 Load/Store 指令外，其他指令都只需一个或小于一个的时钟周期就可完成，指令周期短。

⑤ 采用大量通用寄存器：编译器可将变量分配到寄存器中，以减少访存次数。

⑥ 采用组合逻辑电路控制：指令少而规整使得控制器的实现变得简单，可以不用或少用微程序控制。

⑦ 采用优化的编译系统：指令数少有利于编译器的优化。

采用 RISC 技术后，由于指令系统简单，CPU 的控制逻辑大大简化，芯片上可设置更多的通用寄存器，指令系统也可以采用速度较快的硬连线逻辑来实现，且更适合于采用指令流水技术，这些都可以使指令的执行速度进一步提高。指令数量少，固然使编译工作量加大，但由于指令系统中的指令都是精选的，编译时间少，反过来对编译程序的优化又是有利的。

20 世纪 70 年代中期，IBM 公司、斯坦福大学、加州大学伯克利分校等机构先后开始对 RISC 技术进行研究，其成果分别用于 IBM、SUN、MIPS 等公司的产品中，如美国加州伯克利大学的 RISC I，斯坦福大学的 MIPS，IBM 公司的 IBM801 相继宣告完成，这些机器被称为第一代 RISC 机。到 20 世纪 80 年代中期，RISC 技术蓬勃发展，广泛使用，并以每年翻番的速度发展，先后出现了 Power PC、MIPS、Sun SPARC、Compaq Alpha 等高性能 RISC 芯片以及相应的计算机。这时不少 RISC 的支持者开始对传统的 CISC 计算机（如 VAX、Intel、IBM 大型机）进行攻击，认为未来的发展非 RISC 莫属。当然，这也遭遇到计算机结构主流派的反对，这种争论延续了数年。

虽然 RISC 技术在性能上有优势，但最终 RISC 机并没有在市场上占优势，反而 Intel 一直保持处理器市场的较大份额，这是为什么呢？其原因主要有两点：第一，因为软件的向后兼容性，许多用户先期花了很多钱投资购买了在 Intel 系列机上开发的软件，如果换成 RISC 机，就意味着所有软件要重新投资；其次，随着处理器速度和芯片密度等的不断提高，RISC 系统也日趋复杂，而 CISC 由于采用了部分 RISC 技术（例如，Intel Pentium 4 中将简单指令直接转换为类 RISC 指令，复杂指令用微码实现），使其性能更加提高。虽然这种混合方案不如纯 RISC 方案速度快，但它却能在保证软件兼容的前提下达到具有较强竞争力的整体性能。

5.3 指令系统实例

指令系统设计是计算机系统设计的核心工作，不同机器的指令集体系结构差异很大，了解不同类型的 ISA，对计算机系统设计工作会有很大帮助。以下简单介绍几种有代表性的指令系统实例。

*5.3.1 Pentium 指令系统

Pentium 处理器是 IA-32 体系结构的典型代表，采用 CISC 风格设计。

1. 指令格式

Pentium 采用可变长指令格式，指令长度为 1～15 字节，由前缀和指令两部分组成。

(1) 前缀。位于指令操作码前，但并不是每条指令都必须有，包括以下 4 类。

① 指令前缀：由 LOCK 前缀和重复操作前缀组成。LOCK 前缀规定是否对共享存储器以独占方式使用；重复操作前缀表示重复操作的类型，有 REP、REPZ、REPE、REPNZ、REPNE 等几种。

② 段前缀：如果有段前缀，则指令采用段前缀指定的段寄存器，而不用缺省段寄存器。

③ 操作数长度前缀：如果有该前缀，则操作数长度将采用它规定的长度。

④ 地址长度前缀：如果有该前缀，则地址长度将采用它规定的而不是缺省的长度。

各类前缀的字节数如图5.6所示。

前缀类型：	指令前缀	段前缀	操作数长度	地址长度
字节数：	0或1	0或1	0或1	0或1

图5.6　Pentium指令前缀的字节数

（2）指令。指出操作类型和操作数或操作数地址，各部分长度和含义如图5.7所示。

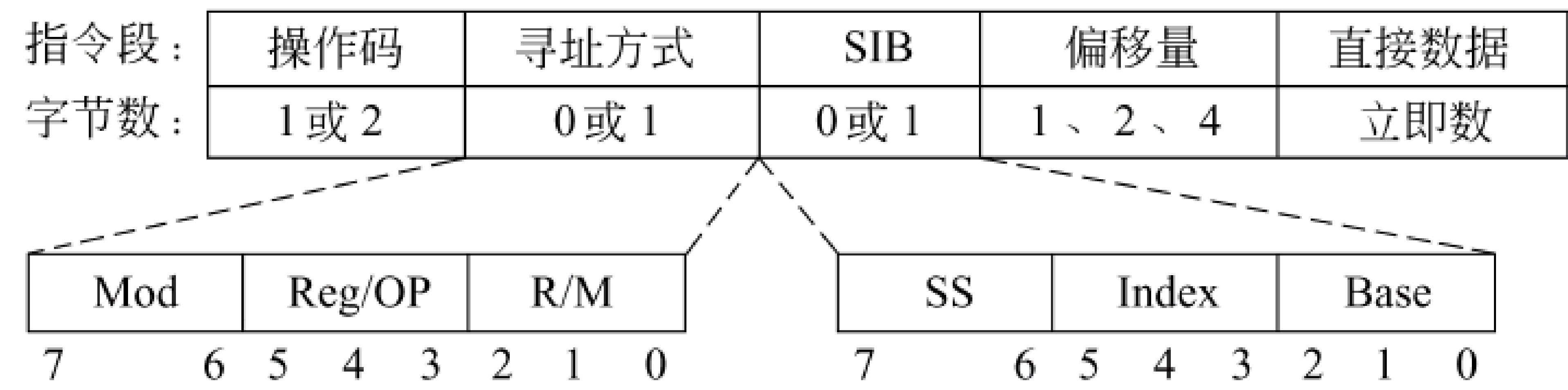


图5.7　Pentium指令格式

指令部分包含以下几个字段。

① 操作码：1～2字节，指定操作性质并给出以下信息：数据长度是字节还是字(16位)；寻址方式字节中Reg/OP字段指定的寄存器是源还是目标；指令中如果有立即数，则是否对它进行符号扩展。

② 寻址方式字节：由Mod、Reg/OP和R/M三个段组成，Mod和R/M联合指定8种寄存器寻址和24种变址寻址方式，Reg/OP指定寄存器号(作为操作数)或3位扩展操作码。

③ SIB：由SS(2位)、Index(3位)、Base(3位)三部分组成。SS指定比例系数(变址寻址方式时，变址寄存器内容要乘以该系数)；Index指定变址寄存器号；Base指定基址寄存器号。

④ 偏移量：指令中如果有偏移量，可以是1、2或4个字节。

⑤ 直接数据：指令中如果有立即数，可以是1、2或4个字节。

2. 寻址方式

操作数来源有三种：立即数、寄存器和存储单元。有32位、16位和8位三种长度的通用寄存器。当操作数为存储单元内容时，需要进行地址转换，Pentium采用段页式存储管理机制，先通过分段方式将虚拟(逻辑)地址转换为线性地址LA，然后再用分页方式将线性地址转换为内存(物理)地址。

指令中必须显式或隐式地给出以下信息。

（1）段寄存器SR(可用段前缀显式给出，也可缺省)。

（2）8/16/32位偏移量A(由偏移量字段显式给出)。

（3）基址寄存器B(由SIB中Base字段显式给出，任意通用寄存器皆可)。

（4）变址寄存器I(由SIB中Index字段显式给出，除ESP外的任意通用寄存器皆可)。

有比例变址和非比例变址。比例变址时，要乘以比例因子S(由SIB中SS字段给出)，其含义为操作数的字节个数。表5.1列出了Pentium的主要寻址方式。

表 5.1　Pentium 的寻址方式

寻　址　方　式	说　　明
立即寻址	指令直接给出操作数
寄存器寻址	指定的寄存器 R 的内容为操作数
位移	LA＝(SR)＋A
基址寻址	LA＝(SR)＋(B)
基址加位移	LA＝(SR)＋(B)＋A
比例变址加位移	LA＝(SR)＋(I)×S＋A
基址加变址加位移	LA＝(SR)＋(B)＋(I)＋A
基址加比例变址加位移	LA＝(SR)＋(B)＋(I)×S＋A
相对寻址	LA＝(PC)＋A

注：LA：线性地址　(X)：X 的内容　SR：段寄存器　PC：程序计数器　R：寄存器
　A：指令中给定地址段的位移量　B：基址寄存器　I：变址寄存器　S：比例系数

Pentium 线性地址 LA 按图 5.8 所示方法形成。首先，根据段寄存器中的段选择符，得到一个段表项，根据其中的段描述符，得到段基址、段长度界限、段存取权限等，再将基址值、比例变址值、偏移量与段基址相加，得到线性地址。可以根据段限和存取权限判断是否“地址越界”和“访问超限”，以实现存储保护。形成线性地址后，再通过分页方式实现从线性地址到物理地址的转换，这个步骤对程序员来讲是透明的。

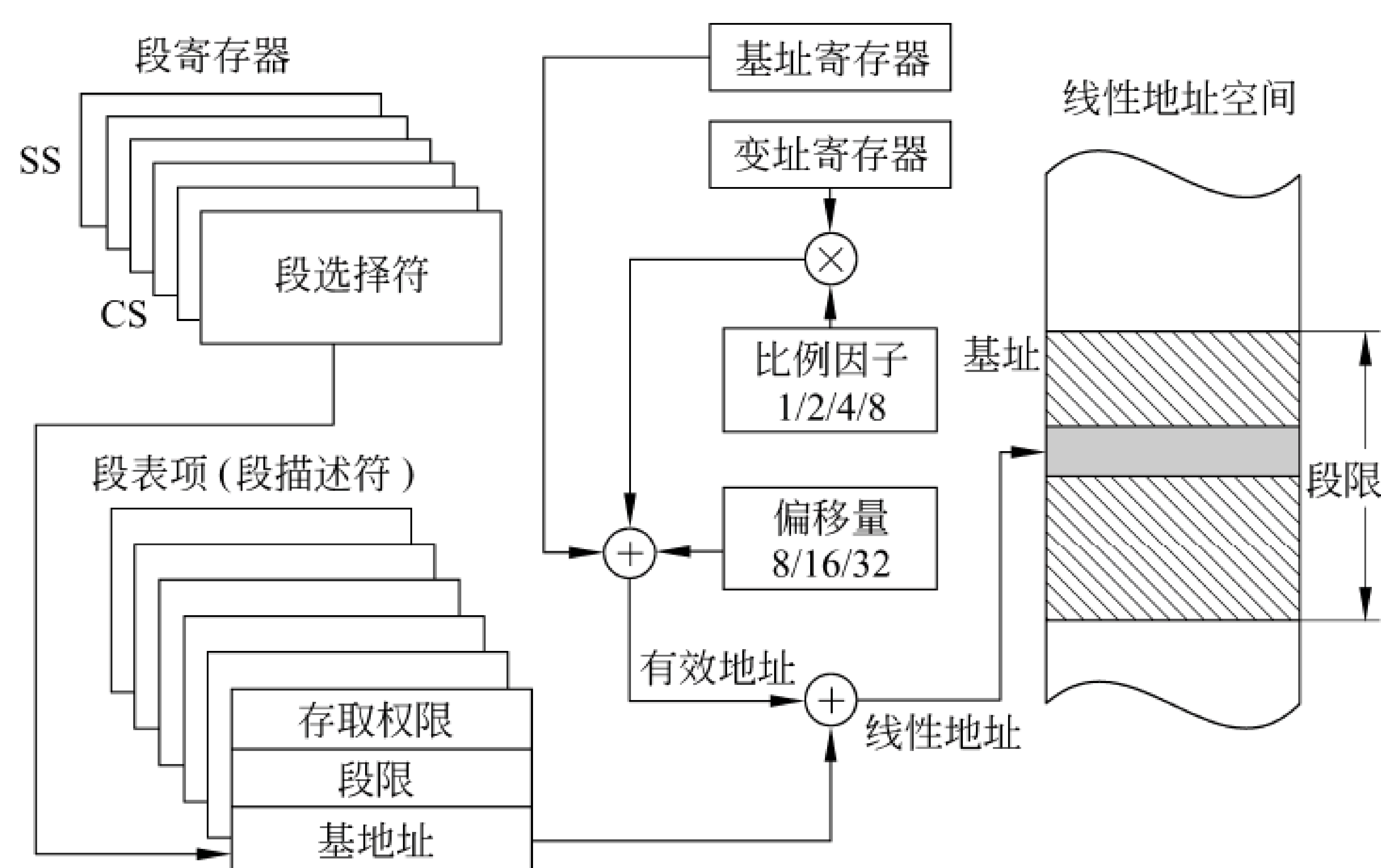


图 5.8　Pentium 处理器线性地址形成过程

此外，Pentium 指令系统中的调用指令自动把返回地址压到栈中，并有专门的入栈(push)和出栈(pop)指令，能自动修改栈指针。运算类指令能直接生成条件码，存放在标志寄存器 EFLAGS 中，而且运算类指令中的一个操作数可以来自主存单元。

*5.3.2 Power PC 指令系统

Power PC 是 RISC 风格处理器，因此，其指令格式和寻址方式较为简单和规整。

1. 指令格式

Power PC 采用 32 位定长指令字结构，指令格式列于表 5.2 中。采用这种规整指令格式，有利于简化指令执行部件的设计。

表 5.2 Power PC 指令格式简表

6 位	5 位	5 位	16 位		
条件转移	选项	CR 位	转移位移量	A	L
条件转移	选项	CR 位	通过计数器或链接寄存器间接		L
无条件转移	直接转移地址				L

(a) 转移指令

6 位	5 位	5 位	5 位		
CR	目标位	源位	源位	与、或、异或等	/

(b) 条件寄存器逻辑指令

6 位	5 位	5 位	11 位		
取数/存数间接	目标寄存器	基址寄存器	位移		
取数/存数间接	目标寄存器	基址寄存器	变址寄存器	大小、符号、更新	/
取数/存数间接	目标寄存器	基址寄存器	位移		XO

(c) 取数/存数指令

6 位	5 位	5 位	← 16 位 →				
算术运算	目标寄存器	源寄存器	源寄存器	O	ADD、SUB 等		R
ADD SUB 等	目标寄存器	源寄存器	有符号立即数				
逻辑运算	目标寄存器	源寄存器	源寄存器	ADD、SUB 等			R
AND、OR 等	目标寄存器	源寄存器	有符号立即数				
环移	目标寄存器	源寄存器	移位量	屏蔽起点	屏蔽终点		R
环移 平移	目标寄存器	源寄存器	源寄存器	移位类型或屏蔽			R
环移	目标寄存器	源寄存器	移位量	屏蔽字	XO	S	R
环移	目标寄存器	源寄存器	移位量	屏蔽字	XO		R
环移	目标寄存器	源寄存器	源寄存器	类型或屏蔽字		S	R

(d) 整数算术/逻辑运算及环移/平移指令

浮点运算	目标寄存器	源寄存器	源寄存器	源寄存器	浮点加等	R

(e) 浮点运算指令

注：A：绝对或 PC 相对　　L：链接至子程序　　O：将溢出标志记录到 XER 中
R：将状态标志记录到 CR1 中　XO：操作码扩展　　S：移位量字段的一部分

所有指令的最高6位都是操作码，有些指令将其他一些位(XO字段)作为操作码扩展。取数/存数、算术运算和逻辑运算指令在操作码后有2～3个5位的寄存器字段，各自选取32个寄存器之一。

转移指令中有一位为链接指示(L)，表示是否将该指令随后一条指令的地址(即返回地址)送入链接寄存器。第一种转移指令中还有一个A标志位，用于指明地址是绝对地址还是相对于PC的地址。CR位用于指明转移条件对应条件寄存器(标志寄存器)中哪一位标志，选项字段用于指定按什么条件转移。

大多数运算类指令(定点算术、浮点算术和逻辑运算)都有一个R标志位，用来表示是否将运算结果的有关标志写入到条件寄存器中，这在条件转移预测时很有用。

浮点运算指令有三个源寄存器字段，大多数情况下只用到两个寄存器，少数指令将两个数相乘然后加上或减去第三个数(即乘加指令和乘减指令)，这类指令主要用于3D图形的坐标转换和矩阵内积计算。

2. 寻址方式

Power PC采用的寻址方式较简单，主要有以下几种。

(1) 取数/存数指令的寻址方式

这类指令主要有两种寻址方式：间接寻址和间接变址寻址。间接寻址的有效地址EA=(BR)+D，BR为基址寄存器，任何一个通用寄存器均可作为基址寄存器使用；D为偏移量，是一个16位带符号数。间接变址寻址的有效地址EA=(BR)+(IR)，IR为变址寄存器，任意一个通用寄存器也均可作为变址寄存器使用。

(2) 转移指令的寻址方式

① 绝对寻址方式：无条件转移和条件转移指令分别给出24位和16位地址，这些地址均扩展为32位后形成转移地址。扩展的方法是，最低端补两个零，高端进行符号扩展。

② 相对寻址方式：如果是无条件转移，指令给出24位地址，将它按前述方法扩展后和PC相加即为转移目标地址；如果是条件转移，指令给出14位地址，按前述方法扩展后和PC相加作为转移目标地址。

③ 间接寻址方式：下条指令的有效地址存放于链接寄存器或计数寄存器中。

(3) 算术指令的寻址方式

整数算术指令可采用寄存器寻址或立即寻址，立即数是16位有符号数。浮点算术指令只可采用寄存器寻址。

*5.3.3 MMX和SIMD指令技术

在多媒体应用中，图形、图像、视频和音频处理过程存在大量具有共同特征的操作。以下列出几种有代表性的操作类型。

① 短整数类型的并行操作，例如8位图像像素和16位音频信号。

② 频繁的乘法累加，例如FIR滤波、矩阵运算。

③ 短数据的高度循环运算，例如快速傅里叶变换FFT、离散余弦变换DCT。

④ 计算密集型算法，例如三维图形、视频压缩。

⑤ 高度并行操作，例如图像处理。

为提高上述共性操作运算能力，Intel在IA-32指令系统基础上，设计了一套新增指令

集,并对 CPU 内部结构进行了扩充与改进,称为 MMX(MultiMedia eXtensions)技术。MMX 指令于 1997 年首次运用于 P54C Pentium 处理器,称之为多能奔腾,共有 57 条指令。MMX 技术包含以下几个方面。

(1) 引入新的数据类型和通用寄存器

MMX 技术的主要数据类型为定点紧缩(Packed)整数,它定义了以下 4 种新的 64 位数据类型。

① 紧缩字节:紧缩成 8 个字节;

② 紧缩字:紧缩成 4 个字(每字 16 位);

③ 紧缩双字:紧缩成两个双字;

④ 四字:一个 64 位字。

为便于 MMX 指令对上述数据类型进行操作,CPU 中新增 8 个 64 位通用寄存器 MX0~MX7。这些寄存器可用来实现数据运算,但不能用于存储器寻址。

(2) 采用 SIMD(Single Instruction Multi Data)技术

单条指令同时并行处理多个数据元素,如 8 个字节,或 4 个字,或两个双字,或一个 64 位字,这对提高运算速度非常有利。例如,一条指令可以完成图像中 8 个像素的并行操作。

(3) 引入饱和(Saturation)运算

在上述四种 64 位数据类型的运算过程中,引入两类运算模式:环绕运算和饱和运算。

在环绕运算(或称非饱和运算)中,上溢或下溢的结果被截取,即进位被丢掉,仅保留低有效位的值。例如 F3H+1DH=10H。在多媒体处理中,这种运算模式有时会产生问题。例如,对于图像的插值运算,若 a 点亮度值为 F3H,b 点亮度值为 1DH,则根据环绕运算模式,其线性插值的结果为 10H/2=08H。新插入点的亮度值比 b 点还低,这显然不符合线性插值的要求,因此,在这种情况下不能用常规的环绕运算模式。为此,引入饱和运算,其运算规则是当发生上溢或下溢时,其运算结果取各类数据值域的最大值或最小值。例如,对于上述图像的插值运算,其数据类型为无符号数,8 位无符号数的最大值为 FFH,因此,采用无符号数饱和运算模式计算 F3H+1DH 时,因为加法运算发生上溢,所以最终结果为 FFH,其线性插值的结果为 FFH/2=7FH,显然,该亮度值作为插值结果比较合理。

随着网络、通信、语音、图形、图像、动画和音/视频等多媒体处理软件对处理器性能越来越高的要求,Intel 在多能奔腾以后的处理器中加入了更多流式 SIMD 扩展(Stream SIMD Extension,SSE)指令集,包括 SSE、SSE2、SSE3、SSSE3 等。

5.4 程序的机器级表示

不管用什么高级语言编写的源程序最终都必须翻译(汇编、解释或编译)成以指令形式表示的机器语言,才能在计算机上运行。本节简单介绍高级语言源程序转换为机器代码过程中涉及的一些基本问题。为方便起见,本节选择具体语言进行说明,高级语言和机器语言分别选用 C 语言和 MIPS 指令系统。其他情况下,其基本原理不变。

*5.4.1 MIPS 汇编语言和机器语言

机器语言程序是一个由若干条指令组成的序列。从前面对指令格式的介绍可以知道,

每条指令由若干字段组成，每个字段都是一串由0、1组成的二进制数字序列。所以，程序员要读懂一个机器语言程序很费劲，也很难用机器语言直接编写程序。

为了能直观地表示机器语言程序，引入了一种与机器语言一一对应的符号化表示语言，称为汇编语言。汇编语言中，用容易记忆的英文单词或缩写来表示指令操作码的含义，用标号、变量名称、寄存器名称、常数等表示操作数或地址码。这些英文单词或缩写、标号、变量名称等都被称为助记符。以下简要介绍MIPS指令系统和MIPS汇编语言。

1. MIPS指令中数据的表示

对于寄存器数据，MIPS提供了32个32位通用寄存器，寄存器编号占5位，各寄存器的名称、编号和功能见表5.3。

表5.3 MIPS通用寄存器

名　称	编　号	功　　能
zero	0	恒为0
at	1	为汇编程序保留
v0～v1	2～3	过程调用返回值
a0～a3	4～7	过程调用参数
t0～t7	8～15	临时变量，在被调用过程无须保存
s0～s7	16～23	在被调用过程需保存
t8～t9	24～25	临时变量，在被调用过程无须保存
k0～k1	26～27	为OS保留
gp	28	全局指针
sp	29	栈指针
fp	30	帧指针
ra	31	过程调用返回地址

寄存器的汇编表示以$开始，可以使用名称，也可以使用编号（$0～$31）。

MIPS还提供了32个32位的单精度浮点寄存器，用汇编符号（f0～f31）表示。它们可配对成16个64位浮点寄存器，用来表示64位双精度浮点数。

另外，MIPS中提供了两个乘商寄存器Hi和Lo，它们是专用寄存器，无须程序员在指令中明显给出。用32位的Hi和Lo可实现64位寄存器。在执行乘法运算时，Hi和Lo联合用来存放64位乘积，而在执行除法运算时，最终的余数存放在Hi中，商在Lo中。

MIPS中用程序计数器PC指出下条指令的地址，它也是专用寄存器。

MIPS的存储器按字节编址。对于存储器数据，其操作数地址为32位，通过一个32位寄存器的内容加16位偏移量得到，16位偏移量是带符号数，故可访问的地址空间大小为2^{32}字节。采用大端方式（Big Endian）存放数据，数据要求按字边界对齐。只能通过Load/Store指令访问存储器数据。

对于立即操作数，指令中给出的位数为16位，指令执行时，需要将其进行符号扩展或0扩展，变成32位操作数后才能参加运算。

2. MIPS 指令格式和寻址方式

MIPS 是典型的 RISC 处理器，采用 32 位定长指令字，操作码字段也是固定长度，没有专门的寻址方式字段，由指令格式确定各操作数的寻址方式。

指令格式只有三种，如图 5.9 所示。



图 5.9　MIPS 指令格式

R-型指令是 RR 型指令，其操作码 OP 为“000000”，操作类型由 func 字段指定，若是双目运算类指令，则 rs 和 rt 的内容分别作为第一和第二源操作数，结果送 rd；若是移位指令，则对 rt 的内容进行移位，结果送 rd，所移位数由 shamt 字段给出。因为一条指令需要左移或右移若干位，所以 MIPS 中移位指令多用桶形移位器实现以提高速度。R 型指令的寻址方式只有一种，就是寄存器寻址。

I-型指令是立即数型指令，若是双目运算类指令，则将 rs 的内容和立即数分别作为第一和第二源操作数，结果送 rt；若是 Load/Store 指令，则将 rs 的内容和立即数符号扩展后的内容相加作为存储单元地址，Load 指令将内存单元内容送 rt，Store 指令将 rt 内容送内存单元；若是条件转移（分支）指令，则对 rs 和 rt 内容进行指定的运算，根据运算的结果，决定是否转到转移目标地址处执行，转移目标地址通过相对寻址方式得到，即将 PC 的内容和立即数符号扩展后的内容相加得到。由此可知，I 型指令的寻址方式有 4 种，就是寄存器寻址、立即数寻址、相对寻址、基址或变址寻址。

J-型指令主要是无条件跳转指令，指令中给出的是 26 位直接地址，只要将当前 PC 的高 4 位拼上 26 位直接地址，最后添两个“0”就可以得到 32 位的跳转目标地址。J 型指令的寻址方式只有一种，就是变通的直接寻址。

例 5.2　为什么 J-型指令中的跳转目标地址最后两位要添“0”，如何实现该功能？

答：因为 MIPS 机器采用 32 位定长指令字，占 4 个字节，其存储单元采用字节编址，所以一条指令占 4 个存储单元，因而，指令地址总是 4 的倍数，最后两位总是“0”，无须在指令中明显给出，只要在实现指令功能的数据通路中具有添加“00”的电路即可。

3. MIPS 汇编语言

表 5.4 和表 5.5 分别是 MIPS 汇编语言和机器代码示例列表。表 5.4 中列出了常用的 5 类指令：算术运算、存储访问、逻辑运算、条件分支、无条件转移。每类中给出最具代表性的指令的名称、汇编形式示例、含义和文字说明。表 5.5 中给出了常用指令的机器代码示例，分别包括操作码汇编助记符、指令格式类型、指令各字段的十进制值和对应的汇编表示。

表 5.4　MIPS 汇编语言示例列表

类别	指令名称	汇编举例	含义	备注
算术运算	add	add $s1,$s2,$s3	$s1=$s2+$s3	三个寄存器操作数
	subtract	sub $s1,$s2,$s3	$s1=$s2−$s3	三个寄存器操作数
存储访问	load word	lw $s1,100($s2)	$s1=Memory[$s2+100]	从内存取一个字到寄存器
	store word	sw $s1,100($s2)	Memory[$s2+100]=$s1	从寄存器存一个字到内存
逻辑运算	and	and $s1,$s2,$s3	$s1=$s2 & $s3	三个寄存器操作数,按位与
	or	or $s1,$s2,$s3	$s1=$s2	$s3	三个寄存器操作数,按位或
	nor	nor $s1,$s2,$s3	$s1=~($s2 | $s3)	三个寄存器操作数,按位或非
	and immediate	andi $s1,$s2,100	$s1=$s2 & 100	寄存器和常数,按位与
	or immediate	ori $s1,$s2,100	$s1=$s2	100	寄存器和常数,按位或
	shift left logical	sll $s1,$s2,10	$s1=$s2≪10	按常数对寄存器逻辑左移
	shift right logical	srl $s1,$s2,10	$s1=$s2≫10	按常数对寄存器逻辑右移
条件分支	branch on equal	beq $s1,$s2,L	if($s1==$s2) go to L	相等则转移
	branch on not equal	bne $s1,$s2,L	if($s1!=$s2) go to L	不相等则转移
	set on less than	slt $s1,$s2,$s3	if($s2<$s3) $s1=1; else $s1=0	小于则置寄存器为1,否则为0,用于后续指令判0
	set on less than immediate	slti $s1,$s2,100	if($s2<100) $s1=1; else $s1=0	小于常数则置寄存器为1,否则为0,用于后续指令判0
无条件跳转	jump	j L	go to L	直接跳转至目标地址
	jump register	jr $ra	go to $ra	过程返回
	jump and link	jal L	$ra=PC+4; go to L	过程调用

表 5.5　MIPS 机器代码示例列表

指令	格式	指令举例(机器代码)						汇编表示
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000
jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					jal 10000

从这两个表中可明显看出机器代码和汇编表示的一一对应关系。根据指令代码和汇编表示之间的对应表(称为指令解码表),可以很容易地实现两者的转换。从汇编表示转换为机器代码的过程称为“汇编”,从机器代码转换为汇编表示的过程称为“反汇编”。

例 5.3 若从 MIPS 指令机器代码与汇编表示对应表中查出操作码(OP 字段)“000000”对应 R-型指令,又从 R-型指令解码表中查到功能码(func 字段)“100000”对应“add”指令。回答以下问题。

(1) 汇编表示“add $t0,$s1,$s2”对应的 MIPS 指令的机器代码是什么?

(2) 假定一条 MIPS 指令的二进制机器代码表示为 0000 0000 1010 1111 1000 0000 0010 0000,则该指令对应的 MIPS 汇编表示形式是什么?

解:汇编和反汇编过程依赖于解码表进行,只要通过查表就可实现指令的机器代码和汇编表示之间的转换。

(1) 由表 5.5 可知,add 指令是 R-型指令,对应的 OP 字段为 0,即二进制 000000,func 字段为 32,即二进制 100000,shamt(移位位数字段)为 0;由表 5.3 得知,$t0、$s1 和 $s2 的编号分别为 8、17 和 18,该指令各字段的值如图 5.10 所示。

31 26	25 21	20 16	15 11	10 6	5 0
000000	10001	10010	01000	00000	100000
OP	rs	rt	rd	shamt	func

图 5.10 指令各字段分解

因此,汇编表示“add $t0,$s1,$s2”对应的指令机器代码是 0000 0010 0011 0010 0100 0000 0010 0000。

(2) 指令的前 6 位操作码为 000000,是一条 R-型指令,按照 R-型指令的格式,指令分解为如图 5.11 所示的 6 个字段,得到 rs=00101,rt=01111,rd=10000,shamt=00000,func=100000。

31 26	25 21	20 16	15 11	10 6	5 0
000000	00101	01111	10000	00000	100000
OP	rs	rt	rd	shamt	func

图 5.11 指令各字段分解

由表 5.5 知是 add 操作,rs、rt、rd 的十进制值分别为 5、15、16,从表 5.3 知,它们分别为 $a1、$t7 和 $s0。故汇编形式为“add $s0,$a1,$t7”或“add $16,$5,$15”。

*5.4.2 选择结构的机器代码表示

过程式程序设计语言提供顺序、选择和循环三种控制结构。选择结构根据判定条件来控制一些语句是否被执行,例如,有 if-then、if-then-else、case(或 switch)、if…goto 等选择语句。对应高级语言中的这些选择语句,在机器语言中提供了各种条件码(标志位)的设置功能以及各种分支(条件转移)指令和无条件转移指令。编译器通过条件码设置指令和各类转移指令来实现程序中选择结构语句。

例 5.4 假定 C 语言赋值语句“f=(g+h)−(i+j);”中变量 i、j、f、g、h 由编译器分别分配给 MIPS 寄存器 $t0~$t4。要求给出该语句编译后的 MIPS 机器代码和汇编表示。

解：只要用三条 R-型指令即可，其中两条 add 指令，一条 sub 指令，所以，三条指令的 OP 字段都为 000000，根据表 5.5 可知，add 指令的 func 字段为 32=100000B，sub 指令的 func 字段为 34=100010B。从表 5.3 可知，寄存器 $t0～$t7 对应 8～15。所以，上述语句对应的 MIPS 机器代码和汇编表示(#后为注释)如下。

```
000000 01011 01100 01101 00000 100000      add $t5, $t3, $t4      # g+h
000000 01000 01001 01110 00000 100000      add $t6, $t0, $t1      # i+j
000000 01101 01110 01010 00000 100010      sub $t2, $t5, $t6      # f= (g+h)- (i+j)
```

例 5.5　以下是一个 C 程序段：if (i==j) f=g+h; else f=g−h; 假定 i、j、f、g、h 由编译器分别分配给 MIPS 寄存器 $s1，$s2，$s3，$s4，$s5。要求给出编译后的 MIPS 汇编表示。

解：首先要有一条分支指令能根据 i、j 是否相等进行转移，所以，选用表 5.4 中 beq 或 bne 指令。此外，还要有一条无条件转移指令，可选用表 5.5 中的 j 指令。上述程序段对应的 MIPS 汇编表示如下。

```
      bne $s1, $s2, else          # if i≠j, jump to else
      add $s3, $s4, $s5           # f=g+h
      j    exit                   # jump to exit
else: sub $s3, $s4, $s5           # f=g-h
exit:
```

例 5.6　以下是一个 C 程序段：if (i<j) f=g+h; else f=g−h ; 假定 i、j、f、g、h 由编译器分别分配给 MIPS 寄存器 $s1，$s2，$s3，$s4，$s5。要求给出编译后的 MIPS 汇编表示。

解：首先要有一条比较 i、j 大小的指令，该指令能根据比较结果设置标志位，然后用分支指令根据标志位的值进行转移，所以，应选用表 5.4 中的 slt 指令、beq 或 bne 指令。在比较标志位的值时，需要判断是否为 0，此时，用 0 号寄存器 $zero 表示 0。上述程序段对应的 MIPS 汇编表示如下。

```
      slt $s6, $s1, $s2           # if i<j, $s6= 1, else $s6=0
      beq $s6, $zero, else        # if $s6=0, jump to else
      add $s3, $s4, $s5           # f=g+h
      j exit                      # jump to exit
else: sub $s3, $s4, $s5           # f=g-h
exit:
```

*5.4.3　循环结构的机器代码表示

循环结构是指可重复执行的一组语句，例如，有 while、until、for、loop、…、goto loop 等循环语句。分支指令在循环结构中也起重要作用，主要用来判断循环条件是否结束。此外，大多数循环体内需要对数组元素进行处理，因此，需要用到自动变址寻址，如果指令系统不提供自动变址，则编译器需要选用对变址器进行增量的指令来使每次循环按顺序取不同的数组元素。

例 5.7 以下是一个 C 程序段：

```
while (i!=k) {
    x=x+A[i];
    i=i+1;
}
```

假定 x、i、k 由编译器分别分配给 MIPS 寄存器 $ s1，$ s2，$ s3，数组 A 的每个元素为一个 32 位字，首地址存放在 $ s5 中，要求给出编译后的 MIPS 汇编表示。

解：循环体内有一个数组元素的访问，首先要计算每次循环中数组元素 A[i]的地址，它应等于 A 的首地址加上偏移量，因为是字数组，所以偏移量等于 i×4，可以用乘法指令实现，也可以用加法指令（两次加倍）或移位指令（左移两位）来实现×4。从前面第 3 章介绍的运算算法来看，乘法指令所需的时间最长，所以一般不用乘法指令实现×4 操作。从内存取数组元素用指令 lw 实现。

循环开始时，先用分支指令 beq 判断循环结束条件，以便在循环结束条件满足时跳出循环体。循环结束后的第一条指令用一个标号 Exit 标识；循环最后要有一条无条件转移指令 j，以转到循环体的开始，循环体内第一条指令的标号为 Loop。

MIPS 没有自动变址寻址，所以用一条显式加法指令 addi 实现数组下标增量。

编译后的 MIPS 汇编表示如下：

```
Loop:  beq $s3, $s2, Exit
       add $s7, $s2, $s2          # i×2
       add $s7, $s7, $s7          # i×4
       add $s7, $s7, $s5
       lw $s6, 0($s7)             # $6=A[i]
       add $s1, $s1, $s6          # x=x+A[i]
       addi $s2, $s2, 1           # i+1
       j Loop
Exit:  …
```

* 5.4.4 过程调用的机器代码表示

子程序的使用有助于提高程序的可读性，并有利于代码重用，它是程序员进行模块化编程的重要手段。子程序的使用主要是通过过程或函数调用实现，为叙述方便起见，本教材将过程（调用）和函数（调用）统称为过程（调用）。过程允许程序员使用参数将过程与其他程序和数据分离，调用过程只要传送输入参数给被调用过程，最后再由被调用过程返回结果参数给调用过程即可。引入过程使得每个程序员只需要关注本模块中函数或过程的编写任务。

将整个程序分成若干模块后，编译器对每个模块分别编译。为了彼此统一，并能配合操作系统工作，编译的模块代码之间必须遵循一些调用接口约定，这些约定由编译器强制执行，汇编程序员也必须强制按照这些约定执行，包括寄存器的使用、堆栈建立和参数传递等。

1. MIPS 中用于过程调用的指令

调用指令是一种无条件转移指令，在 MIPS 中称为跳转并链接（jump and link）指令，指

令名称为 jal，采用 J-型格式，具有两个功能。(1)保存下条指令地址到 31 号寄存器；(2)跳转到指定地址处执行。其汇编形式和指令格式分别参见表 5.4 和表 5.5。例如，指令“jal 10000”的功能为 $31=PC+4；go to 10000。

返回指令也是一种无条件转移指令，在 MIPS 中称为寄存器跳转(jump register)指令，指令名称为 jr，采用 R-型格式，其功能为跳转到寄存器指定的地址处执行。其汇编形式和指令格式分别参见表 5.4 和表 5.5。例如，指令“jr $31”的功能为转到调用程序的返回地址(在 $31 中保存)处执行。jr 指令中的寄存器也可以是除 $31 以外的其他寄存器，所以，jr 指令也可用于 switch 或 case 语句中的跳转执行。

2. 过程调用时 MIPS 寄存器的使用约定

假定过程 P 调用过程 Q，则过程调用的执行步骤如下：

(1) P 将入口参数放到 Q 能访问到的地方；

(2) P 将返回地址存到特定的地方，然后将控制转移到 Q；

(3) Q 为 P 保存现场，并为自己的局部变量分配空间；

(4) 执行过程 Q；

(5) Q 将返回结果放到 P 能访问到的地方；

(6) Q 取出返回地址，将控制转移到 P。

从上述执行步骤来看，在过程调用中，需要为入口参数、返回地址、调用过程执行时用到的寄存器、被调用过程中的局部变量、过程返回时的结果等数据找到存放空间。如果有足够的寄存器，最好把这些数据都保存在寄存器中，这样，CPU 执行指令时，可以快速地从寄存器取得这些数据进行处理。但是，用户可见寄存器数量有限；并且它们是所有过程共享的资源，给定时刻只能被一个过程使用；此外，对于过程中使用的局部数组和结构等复杂类型数据也不可能保存在寄存器中。因此，除了寄存器外，还需要有一个专门的存储结构来保留这些数据，这个存储结构就是栈(Stack)，也称为堆栈。那么，上述数据中哪些存放在寄存器，哪些存放在堆栈中呢？寄存器和栈的使用又有哪些规定呢？

尽管硬件对寄存器的用法几乎没有任何规定，CPU 实现指令功能时完全不用考虑寄存器的功能，但在软件实际使用寄存器时还要遵循一定的惯例，使程序员、编译器和操作系统等都按照统一的约定处理。

假定过程 P 调用过程 Q，则 MIPS 程序中对过程调用时寄存器的使用规定如下(参见表 5.3)。

(1) $a0～$a3 用于传递前 4 个非浮点数入口参数，在过程 P 中应先将入口参数送入 $a0～$a3，然后调用 Q。若入口参数超过 4 个，则其余参数保存到栈中。

(2) $v0～$v1 用于传递从 Q 返回的非浮点数返回参数，在过程 Q 中应先将返回参数送入 $v0～$v1 再返回 P。

(3) $ra 用于存放返回地址，由调用指令(jal)自动将返回地址送入 $ra。

(4) $s0～$s7 在过程 P 中原来的老值从过程 Q 返回后可被 P 继续使用，因此，若在过程 Q 中使用这些寄存器，则必须先将其保存到栈后才能使用，并在返回 P 前恢复，因此，它们被称为保存寄存器。

(5) $t0～$t9 的值从过程 Q 返回后在 P 中不再需要使用，若需要则由 P 自己保存，因此，在过程 Q 中不需对其保存，可以自由使用，因此，它们被称为临时寄存器。

(6) ＄a0～＄a3 的值从过程 Q 返回后在 P 中也不再需要使用，若需要则由 P 自己保存，因此，过程 Q 不需要为过程 P 对其进行保存。

3. MIPS 中的栈和栈帧

上文提到，过程调用时的一些数据除了可存放到寄存器外，还有一些数据被存放到栈中。MIPS 中有一个专门的栈指针寄存器＄sp，用来指示栈顶元素，栈中每个元素的长度为 32 位，没有专门的入栈指令（push）和出栈指令（pop）。入栈、出栈操作用 sw/lw 指令实现，因而不能自动进行栈指针调整，需用 addi 指令调整＄sp 的值。

MIPS 中，栈从高地址向低地址方向“增长”，而取数/存数是从低地址向高地址方向进行（MIPS 采用大端方式），每入栈一个字，则＄sp－4→＄sp，每出栈一个字，则＄sp＋4→＄sp。

例 5.8 假定将返回地址＄ra 和参数＄a0 保存到栈中，写出其指令序列，并画图说明＄ra 和＄a0 在栈中的位置。

解：假定栈指针寄存器＄sp 指向栈顶，返回地址＄ra 和参数＄a0 从栈顶处开始存放，其存放位置如图 5.12 所示。

在栈中保存信息的指令序列如下：

```
addi  $sp, $sp,-8
sw    $ra, 4($sp)
sw    $a0, 0($sp)
```

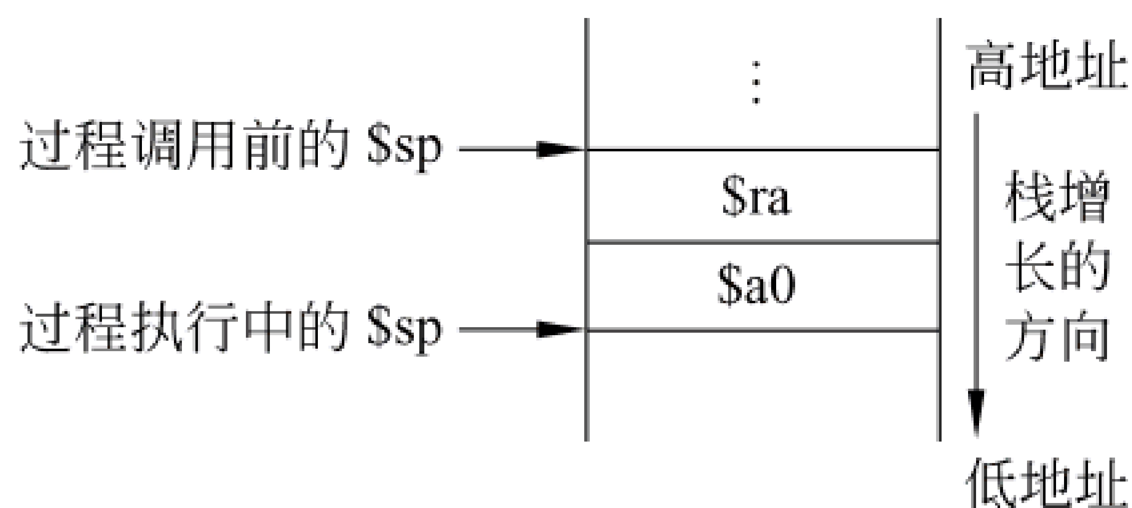


图 5.12 栈中数据的存放

每个过程都有自己的栈区，称为栈帧（Stack Frame），因此，一个栈由若干栈帧组成，每个栈帧用专门的帧指针寄存器指定起始位置，MIPS 中的帧指针寄存器是＄fp。当前栈帧范围在帧指针＄fp 和栈指针＄sp 指向区域之间。过程执行时，由于不断有数据入栈，所以栈指针会动态移动，而帧指针可以固定不变。对程序来说，用固定的帧指针来访问变量要比用变化的栈指针方便得多，也不易出错，因此，在一个过程内对栈中信息的访问大多通过帧指针进行。但是，如果当前过程的栈帧（即当前栈帧）中没有局部变量，则编译器大多不设置和恢复帧指针，以减少时空开销。当需要使用帧指针＄fp 时，通常以过程调用时的栈指针＄sp 或＄sp－4 作为其初始值，这样，＄fp 总是指向当前栈帧前一个字或当前栈帧第一个字的起始位置。

假定过程 P 调用过程 Q，则在调用过程 P 中入栈保存的信息称为调用者保存信息，存放在过程 P 的栈帧中；在被调用过程 Q 中入栈保存的信息称为被调用者保存信息，存放在 Q 的栈帧中。图 5.13 给出了在过程调用前、调用中和调用后的 MIPS 用户栈的变化状态。

如图 5.13(a)所示，在调用过程中遇到新的一个过程调用时，调用过程根据需要确定是否将临时寄存器或参数寄存器保存到自己的栈帧（调用过程栈帧）中，同时，对于浮点数参数和超过 4 个的其余非浮点数参数也要保存到自己的栈帧中，然后转入被调用过程。如图 5.13(b)所示，在被调用过程中，需要时需将帧指针＄fp 设置为＄sp－4（也可设置为＄sp），在＄fp 和＄sp 指向的区间之间的是当前栈帧。如果当前过程是非叶子过程，则返回地址入栈保存；若在过程中用到保留寄存器，则将它们入栈保存；然后根据过程中局部数组

或结构等数据定义情况，对局部变量进行入栈保存；若$fp有被破坏的情况（如嵌套调用）发生，还需将$fp保留到当前栈帧中；如果是递归调用，则所有输入参数都需要入栈保存。被调用过程执行结束返回前，必须释放局部变量占用的栈区，并恢复保存的各个寄存器，最后可根据$fp的值恢复进入被调用过程时的栈指针$sp。这样，在回到调用程序后，栈中状态和过程调用前一样，如图5.13(c)所示。

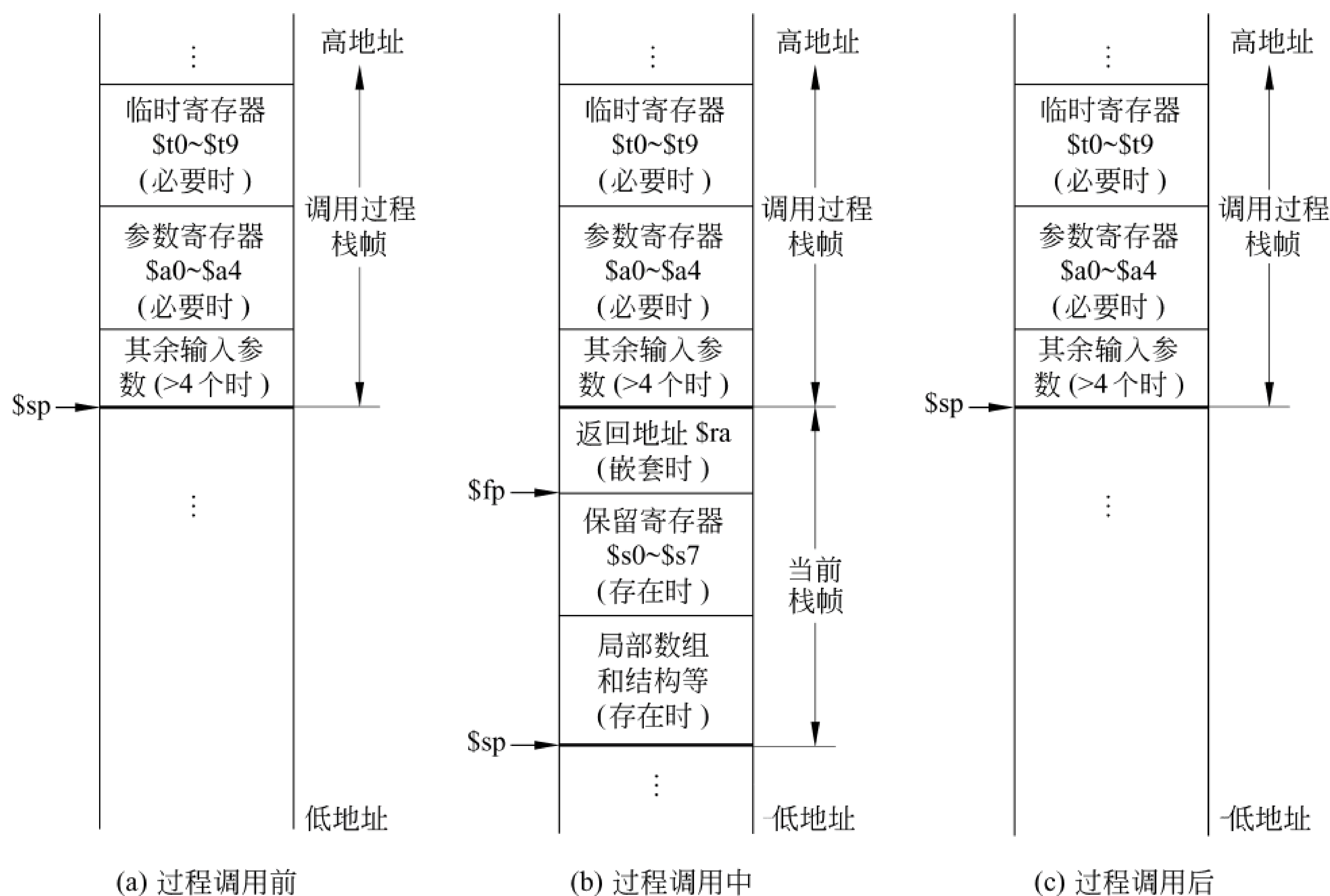


图5.13　过程调用时MIPS中的栈和栈帧的变化

4. MIPS过程调用协议

在程序执行过程中，每调用一次过程，都会在栈中生成一个对应的新栈帧，而在执行返回指令前对应的栈帧在栈中都已被释放。栈帧的生成和释放方式可以有多种方式，但不管采用什么方式，调用程序和被调用程序都必须遵循一定的步骤。

以下步骤是大多数MIPS系统采用的过程调用协议。

(1) 调用程序P在过程调用前的执行步骤

① 将前4个参数送到$a0～$a3，其他参数压入当前栈帧。

② 若P在返回后还要用到$a0～$a3和$t0～$t9中某些寄存器，则需将这些寄存器压到当前栈帧中。

③ 执行jal指令，以将返回地址保存到$ra中，并将控制转移到被调用程序。

(2) 被调用程序Q中的执行步骤

由三段组成：开始段、本体段(过程体)和结尾段。本体段进行具体处理。

开始段主要进行栈帧生成、寄存器保存和局部变量空间申请。其处理步骤如下：

① 通过调整栈指针$sp来申请栈帧，即将$sp的值减去栈帧大小。

② 若Q中用到$s0～$s7中的某些寄存器，则需将这些寄存器压入当前栈帧。

③ 若 Q 需调用其他过程，则将 \$ ra 和帧指针 \$ fp 压入当前栈帧。

④ 若 Q 中的局部变量发生寄存器溢出（即寄存器不够分配），则局部变量在 Q 的栈帧中分配空间；若有像数组和结构之类的复杂类型局部变量，则在当前栈帧中分配空间。

⑤ 设置帧指针 \$ fp，其值为当前栈指针 \$ sp 加栈帧大小再减 4。

由此可见，栈帧大小应至少等于上述②、③、④三个步骤中用到的存储单元的总和。

结尾段主要进行寄存器恢复、栈帧释放，并返回到调用程序。其处理步骤如下：

① 若保存了 \$ s0～\$ s7 中某些寄存器值，则将这些值从当前栈帧中恢复到寄存器。

② 若保存了返回地址和帧指针，则将它们分别恢复到寄存器 \$ ra 和 \$ fp 中。

③ 调整栈指针 \$ sp 以释放栈帧，即将 \$ sp 的值加上栈帧大小，或将 \$ fp 的值送 \$ sp。

④ 用返回指令“jr \$ ra”将控制权返还给调用程序。

例 5.9 写出以下 C 语言过程对应的 MIPS 汇编表示。

```
void swap(int v[ ], int k)
{
    int temp;
    temp=v[k];
    v[k]=v[k+1];
    v[k+1]=temp;
}
```

解：swap 子程序不是主程序（main 函数），因此是一个被调用过程，但它不再调用其他过程，所以是个叶子过程。

按照调用协议，调用 swap 过程的程序已将参数 v 和 k 分别放在参数寄存器 \$ a0 和 \$ a1 中。参数 v 是一个数组的指针。假定在 swap 过程体中先使用临时寄存器 \$ t0～\$ t9，不够时再使用保存寄存器 \$ s0～\$ s7，局部变量 temp 分配在寄存器 \$ t0 中。如果临时寄存器够用的话，则不需要在栈帧中保存调用程序的现场，即不需将 \$ s0～\$ s7 的值保存在栈帧中。

按照上述 MIPS 过程调用协议，开始段无须保存任何寄存器的值，也无须进行局部变量分配，因为是叶子过程，故无须保存返回地址和帧指针，由此可见 swap 对应的栈帧为空；结尾段直接返回即可，swap 的汇编表示如下：

```
swap: sll   $t1, $a1, 2      #k<<2, multiply k by 4
      add   $t1, $t1, $a0    #address of v[k]
      lw    $t0, 0($t1)      #load v[k]
      lw    $t2, 4($t1)      #load v[k+1]
      sw    $t2, 0($t1)      #store v[k+1] into v[k]
      sw    $t0, 4($t1)      #store old v[k] into v[k+1]
      jr    $31              #return to caller
```

例 5.10 以下是三个 C 语言过程，假定过程 set_array 第一个被调用，全局变量 i 分配到寄存器 \$ s0 中。要求写出每个过程对应的 MIPS 汇编表示，并画出每个过程调用前、后栈中的状态以及帧指针和栈指针的位置。

```
int i;
```

```
void set_array (int num)
{
    int array[10];
    for (i=0; i<10; i++) {
        array[i]=compare (num, i);
    }
}

int compare (int a, int b)
{
    if (sub (a, b)>=0)
        return 1;
    else
        return 0;
}

int sub (int a, int b)
{
    return a-b;
}
```

解：程序由三个过程组成，全局静态变量有一个 i，假定分配给 \$ s0。

为了尽量减少指令条数，并减少访问内存次数。在每个过程的过程体中一般先使用临时寄存器 \$ t0～\$ t9，临时寄存器不够或者某个值在调用过程返回后还需要用，就使用保存寄存器 \$ s0～\$ s7。

MIPS 指令系统中没有寄存器传送指令，为了提高汇编表示的可读性，引入一条伪指令 move 来表示寄存器传送，汇编器将其转换为具有相同功能的机器指令。伪指令“move \$ t0，\$ s0”对应的机器指令为“add \$ t0，\$ zero，\$ s0”。

（1）过程 set_array。入口参数为 num，没有返回参数，有一个局部数组，被调用过程为 compare，因此，其栈帧中除了保留所用的保存寄存器外，必须保留返回地址（是否保存 \$ fp 要看具体情况，如果确保后面都不用到 \$ fp，则可以不保存，但为了保证 \$ fp 的值不被后面的过程覆盖，通常情况下，应该保存 \$ fp 的值），并给局部数组预留 $4\times10=40$ 个字节的空间。

从过程体来看，从 compare 返回后还需要用到数组基地址，故将其分配给 \$ s1。因此要用到的保存寄存器有两个：\$ s0 和 \$ s1，其中，\$ s0 中 i 是一个全局变量，所以只有 \$ s1 需要保存在栈中，另外加上返回地址、帧指针和局部数组，其栈帧空间最少为 $4\times3+40=52$B，汇编表示如下。

```
set-array: addi  $sp, $sp, -52        #generate stack frame
           sw    $ra, 48($sp)         #save $ra on stack
           sw    $fp, 44($sp)         #save $fp on stack
           sw    $s1, 40($sp)         #save $s1 on stack
           addi  $fp, $sp, 48         #set $fp
```

```
            move   $s1, $sp               #base address of array
            move   $t0, $a0               #$t0=num
            move   $s0, $zero             #i=0
  for-loop: slti   $t1, $s0, 10           #if i<10, $t1=1; if i>=10, $t1=0
            beq    $t1, $zero, exit       #if $t1=0, jump to exit
            move   $a0, $t0               #$a0=num
            move   $a1, $s0               #$a1=i
            jal    compare                #call compare
            sll    $t1, $s0, 2            #i×4
            add    $t1, $s1, $t1          #$t1=array[i]
            sw     $v0, 0($t1)            #store result to array[i]
            addi   $s0, $s0, 1            #i=i+1
            j      for-loop
      exit: lw     $ra, 48($sp)           #restore $ra
            lw     $fp, 44($sp)           #restore $fp
            lw     $s1, 40($sp)           #restore $s1
            addi   $sp, $sp, 52           #free stack frame
            jr     $ra                    #return to caller
```

(2) 过程 compare。入口参数为 a 和 b,有一个返回参数,没有局部变量,被调用过程为 sub。所以其栈帧中除了保留所用的保存寄存器外,还必须保留返回地址和旧 \$ fp 的值;因为 compare 过程的参数和 sub 过程的入口参数一样,所以在调用 sub 前没有对 \$ a0 和 \$ a1 寄存器送参数。

```
compare:  addi   $sp, $sp, -8
          sw     $ra, 4($sp)
          sw     $fp, 0($sp)             #save $fp on stack
          addi   $fp, $sp, 4             #set $fp
          jal    sub
          slt    $t1, $v0, $zero         #if $v0<0, $t1=1; if $v0>=0, $t1=0
          beq    $t1, $zero, else        #if $t1=0, jump to else
          move   $v0, $zero              #return 0
          j      exit
    else: ori    $v0, $zer0, 1           #return 1
    exit: lw     $fp, 0($sp)
          lw     $ra, 4($sp)
          addi   $sp, $sp, 8
          jr     $ra
```

(3) 过程 sub。入口参数为 a 和 b,有一个返回参数,没有局部变量,是叶子过程,且过程体中没有用到任何保存寄存器。所以栈帧中不需要保留任何信息(如果保留返回地址和 \$ fp 也不会错,但需要额外的指令来执行保存和恢复操作,增加了程序的执行时间,因此,一般不对叶过程的返回地址进行保存)。

```
    sub: sub   $v0, $a0, $a1
```

```
jr   $ra
```

图 5.14 给出了每个过程调用前、后栈中的状态变化以及帧指针和栈指针的位置。

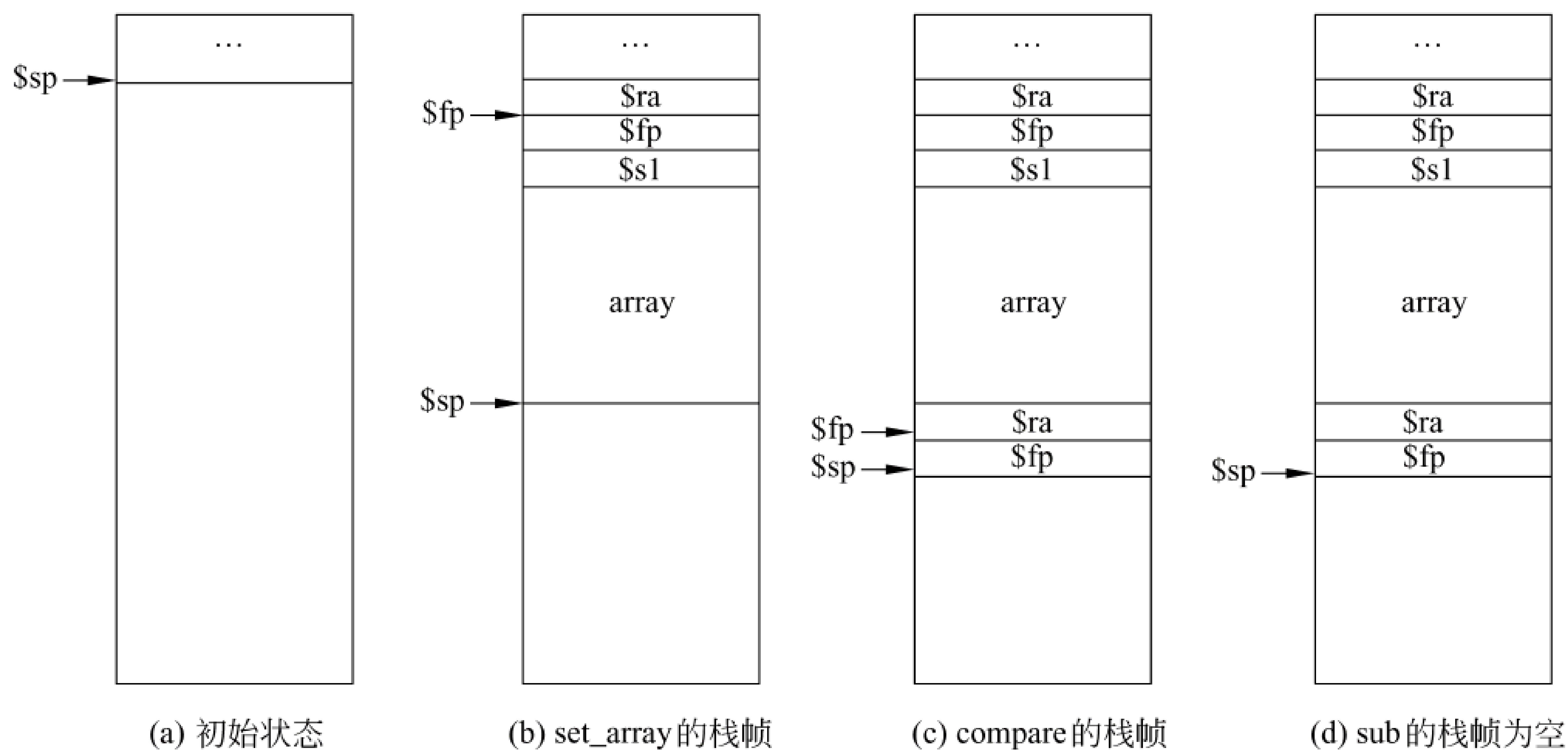


图 5.14　例 5.10 的堆栈中栈指针、帧指针的变化以及所保存的信息

需要说明的是,本例给出的程序是示意性的,实际上该程序没有任何意义。因为过程 set_array 所做的工作就是把比较的结果写到数组 array 中,没有任何返回值,但数组 array 是局部的,当从 set_array 返回后,该过程的栈帧全部被释放,array 中的值也全部无效,所以程序没有做任何工作。此外,从上述例子可以看出,如果全部利用栈指针 $ sp 来访问栈帧是可以实现的,所以,MIPS 中的 30 号寄存器 $ 30 可以作为帧指针 $ fp,也可以在不想利用它来访问栈帧时把它作为保留寄存器 $ s8 使用。

5.5　本章小结

本章通过高级语言与汇编语言程序之间的对应关系,以 MIPS 处理器的指令系统为例介绍了一个指令系统必须具备的基本功能,并就指令系统设计方面的有关内容进行了介绍。包括指令的格式、操作数类型、数据在存储器中的存放方式、寻址方式、操作类型、硬件对过程调用的支持、CISC 和 RISC 技术的比较、指令系统举例和分析设计等,具体总结如下。

- 指令格式
  - 定长指令字、定长操作码:方便指令地址计算、取指和译码。
  - 变长指令字、变长操作码:指令紧凑、程序占空间少。
- 操作类型
  - 数据传送类:数据在寄存器、主存单元、栈顶等之间进行传送。
  - 运算类:各种算术运算和逻辑运算。
  - 字符串处理类:字符串查找、扫描、转换等。
  - I/O 操作类:用于 CPU 与外设接口进行数据/状态/命令信息的交换。
  - 程序流控制类:条件转移、无条件转移、调用、返回等。
  - 系统控制类:启动、停止、自愿访管、自陷、空操作等。

- 操作数类型(以 Pentium 处理器数据类型为例)
  - 序数或指针：8 位、16 位、32 位无符号整数表示。
  - 整数：16 位、32 位、64 位三种补码表示的整数。
  - 实数：IEEE 754 浮点数格式。
  - 十进制数：18 位十进制数，用 80 个二进位表示。
  - 字符串：字节为单位的字符序列，一般用 ASCII 码表示。
- 操作数宽度有多种：对应高级语言中各种简单类型数据长度，如字节、16 位、32 位、64 位等。
- 寻址方式
  - 立即寻址：地址码直接给出操作数本身。
  - 直接寻址：地址码给出操作数所在的内存单元地址。
  - 间接寻址：地址码给出操作数的内存单元地址所在的内存单元地址。
  - 寄存器寻址：地址码给出操作数所在的寄存器编号。
  - 寄存器间接寻址：地址码给出操作数所在单元的地址所在的寄存器编号。
  - 堆栈寻址：操作数约定在堆栈中，总是从栈顶取数或存数。
  - 偏移寻址：用寄存器内容加形式地址得到操作数所在的内存单元地址，包括以下三种。
    - 变址寻址：地址码给出一个形式地址，并且隐式或显式地指定一个寄存器作为变址寄存器，变址寄存器的内容(变址值)和形式地址相加，得到操作数的有效地址，通常用于循环体中对数组元素的访问。
    - 相对寻址：指令中的形式地址给出一个位移量 D，而基准地址由程序计器 PC 提供。即有效地址 EA=(PC)+D，通常用于转移指令中转移目标或公共子程序中的操作数的寻址。
    - 基址寻址：地址码给出的形式地址作为位移量，与基址寄存器的内容相加，得到有效地址。基址寄存器可以在指令中显式指定，也可以用一个专门的基址寄存器。
- 条件码(状态标志)的生成：对应高级语言程序中的选择结构和循环结构，相应的机器代码中需要有条件转移指令，它们根据不同的状态标志来改变程序的执行顺序。通常的状态标志有 CF(进/借位标志)、ZF(零标志)、OF(溢出标志)、SF(符号标志)等。
- 指令系统风格
  - 按地址码指定风格来分
    - 累加器型：其中一个操作数和运算结果都隐含存放在累加器中，指令长度短，但程序的指令条数多，并需要频繁访问存储器。
    - 堆栈型：操作数和结果都隐含在堆栈中，指令长度短，但需频繁访问堆栈，对指令序列的顺序要求严格。
    - 通用寄存器型：操作数明显地指定在通用寄存器中。使用大量通用寄存器，既缩短了指令长度，又减少了访问存储器的次数，所以现代计算机大多采用这种指令设计风格。

  - ▲ 装入/存储型：这种类型指令系统本身是通用寄存器型，同时还规定，对于运算类指令的操作数只能在寄存器中，只有装入(Load)指令和存储(Store)指令才能访问内存。
- ◆ 按指令系统的复杂度来分
  - ▲ CISC(复杂指令系统计算机)：变长指令字，扩展操作码编码，指令格式多，指令条数多，寻址方式多而复杂，因而指令的译码实现复杂，大多用微程序控制器实现。
  - ▲ RISC(精简指令系统计算机)：定长指令字，定长操作码，指令格式少，指令系统中仅含有一些常用指令，因而指令条数少，寻址方式少且简单，指令的译码实现简单，可用硬连线路控制器实现。RISC 处理器中设置大量的通用寄存器，可大大减少存储器访问次数。采用装入/存储(Load/Store)型指令设计风格，因而大部分指令的执行步骤一致、规整，指令的执行适合于采用流水线方式执行。

习 题 5

1. 给出以下概念的解释说明。

(1) 指令	(2) 指令集体系结构 ISA	(3) 操作码	(4) 地址码
(5) 程序计数器 PC	(6) 指令指针 IP	(7) 程序状态字 PSW	(8) 程序状态字寄存器
(9) 标志寄存器	(10) 堆栈	(11) 栈指针 SP	(12) 寻址方式
(13) 有效地址	(14) 立即寻址	(15) 直接寻址	(16) 间接寻址
(17) 寄存器寻址	(18) 寄存器间接寻址	(19) 变址寻址	(20) 变址寄存器
(21) 相对寻址	(22) 基址寻址	(23) 基址寄存器	(24) 堆栈寻址
(25) 通用寄存器 GPR	(26) SIMD 指令	(27) CISC	(28) RISC
(29) 伪指令	(30) 叶过程	(31) 帧指针 fp	(32)栈帧(Stack frame)

2. 简单回答下列问题。

(1) 一条指令中应该显式或隐式地给出哪些信息？

(2) 什么是“汇编”过程？什么是“反汇编”过程？

(3) CPU 如何确定指令中各个操作数的类型、长度以及所在地址？

(4) 哪些寻址方式下的操作数在寄存器中？哪些寻址方式下的操作数在存储器中？

(5) 基址寻址方式和变址寻址方式的作用各是什么？有何相同点和不同点？

(6) 为何分支指令的转移目标地址通常用相对寻址方式？

(7) RISC 处理器的特点有哪些？

(8) CPU 中标志寄存器的功能是什么？有哪几种基本标志？

(9) 转移指令和转子(调用)指令的区别是什么？返回指令是否需要有地址码字段？

3. 假定某计算机中有一条转移指令，采用相对寻址方式，共占两个字节，第一字节是操作码，第二字节是相对位移量(用补码表示)，CPU 每次从内存只能取一个字节。假设执行到某转移指令时 PC 的内容为 200，执行该转移指令后要求转移到 100 开始的一段程序执行，则该转移指令第二字节的内容应该是多少？

4. 假设地址为 1200H 的内存单元中的内容为 12FCH，地址为 12FCH 的内存单元的内容为 38B8H，而 38B8H 单元的内容为 88F9H。说明以下各情况下操作数的有效地址和操作数各是多少？

(1) 操作数采用变址寻址，变址寄存器的内容为 12，指令中给出的形式地址为 1200H。

(2) 操作数采用一次间接寻址，指令中给出的地址码为 1200H。

(3) 操作数采用寄存器间接寻址，指令中给出的寄存器编号为8，8号寄存器的内容为1200H。

5. 通过查资料了解Intel 80x86微处理器和MIPS处理器中各自提供了哪些加法指令，说明每条加法指令的汇编形式、指令格式和功能，并比较加、减运算指令在这两种指令系统中不同的设计方式，包括不同的溢出处理方式。

6. 某计算机指令系统采用定长指令字格式，指令字长16位，每个操作数的地址码长6位。指令分二地址、单地址和零地址三类。若二地址指令有k2条，无地址指令有k0条，则单地址指令最多有多少条？

7. 某计算机字长16位，每次存储器访问宽度16位，CPU中有8个16位通用寄存器。现为该机设计指令系统，要求指令长度为字长的整数倍，至多支持64种不同操作，每个操作数都支持4种寻址方式：立即(I)、寄存器直接(R)、寄存器间接(S)和变址(X)，存储器地址位数和立即数均为16位，任何一个通用寄存器都可作变址寄存器，支持以下7种二地址指令格式：RR型、RI型、RS型、RX型、XI型、SI型、SS型。请设计该指令系统的7种指令格式，给出每种格式的指令长度、各字段所占位数和含义，并说明每种格式指令需要几次存储器访问？

8. 有些计算机提供了专门的指令，能从一个32位寄存器中抽取其中任意一个位串置于另一个寄存器的低位有效位上，并在高位补0，如图5.15所示。MIPS指令系统中没有这样的指令，请写出最短的一个MIPS指令序列来实现这个功能，要求i=5，j=22，操作前后的寄存器分别为$s0和$s2。

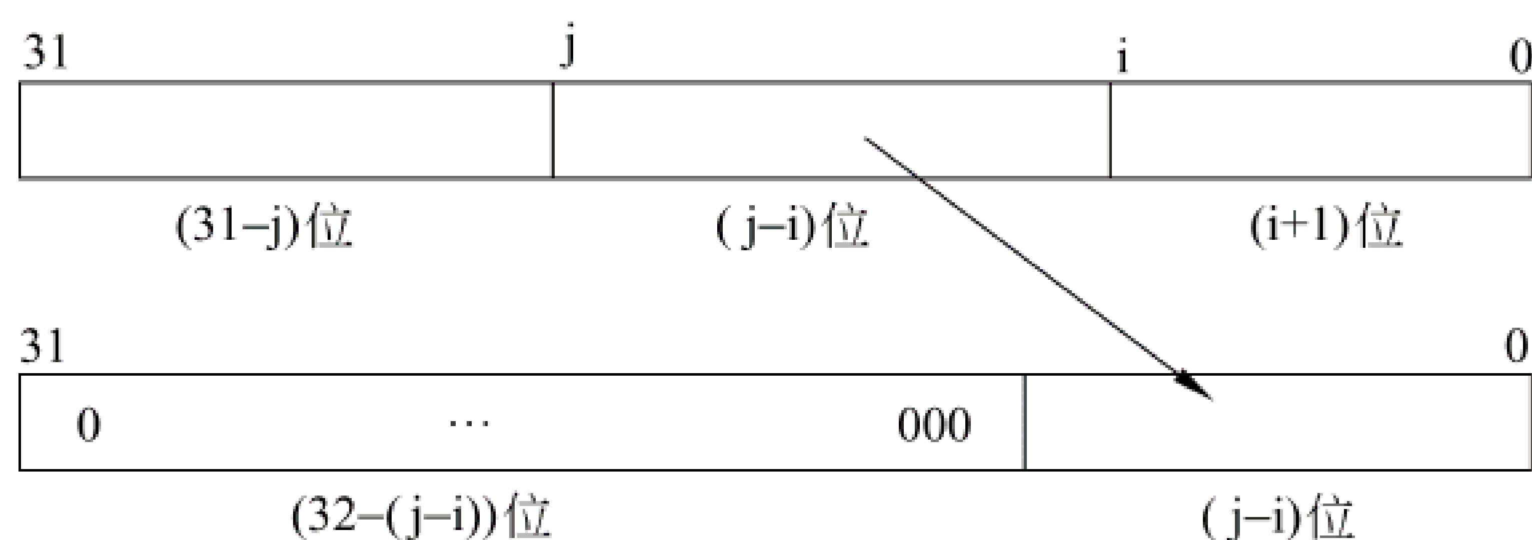


图5.15　题8用图

9. 以下程序段是某个过程对应的指令序列。入口参数int a和int b分别置于$a0和$a1中，返回参数是该过程的结果，置于$v0中。要求为以下MIPS指令序列加注释，并简单说明该过程的功能。

```
        add   $t0, $zero, $zero
loop:   beq   $a1, $zero, finish
        add   $t0, $t0, $a0
        sub   $a1, $a1, 1
        j     loop
finish: addi  $t0, $t0, 100
        add   $v0, $t0, $zero
```

10. 下列指令序列用来对两个数组进行处理，并产生结果存放在$v0中。假定每个数组有2500个字，其数组下标为0到2499。两个数组的基地址分别存放在$a0和$a1中，数组长度分别存放在$a2和$a3中。要求为以下MIPS指令序列加注释，并简单说明该过程的功能。假定该指令序列运行在一个时钟频率为2GHz的处理器上，add、addi和sll指令的CPI为1；lw和bne指令的CPI为2，则最坏情况下运行所需时间是多少秒？

```
        sll   $a2, $a2, 2
        sll   $a3, $a3, 2
        add   $v0, $zero, $zero
        add   $t0, $zero, $zero
outer:  add   $t4, $a0, $t0
        lw    $t4, 0($t4)
```

```
        add  $t1, $zero, $zero
inner:  add  $t3, $a1, $t1
        lw   $t3, 0($t3)
        bne  $t3, $t4, skip
        addi $v0, $v0, 1
 skip:  addi $t1, $t1, 4
        bne  $t1, $a3, inner
        addi $t0, $t0, 4
        bne  $t0, $a2, outer
```

11. 用一条 MIPS 指令或最短的 MIPS 指令序列实现以下 C 语言语句：b=25|a。假定编译器将 a 和 b 分别分配到 \$ t0 和 \$ t1 中。如果把 25 换成 65536，即 b=65536|a，则用 MIPS 指令或指令序列如何实现？

12. 以下程序段是某个过程对应的 MIPS 指令序列，其功能为复制一个存储块数据到另一个存储块中，存储块中每个数据的类型为 float，源数据块和目的数据块的首地址分别存放在 \$ a0 和 \$ a1 中，复制的数据个数存放在 \$ v0 中，作为返回参数返回给调用过程。假定在复制过程中遇到 0 就停止复制，最后一个 0 也需要复制，但不被计数。已知程序段中有多个 Bug，请找出它们并修改之。

```
       addi  $v0, $zero, 0
loop:  lw    $v1, 0($a0)
       sw    $v1, 0($a1)
       addi  $a0, $a0, 4
       addi  $a1, $a1, 4
       beq   $v1, $zero, loop
```

13. 说明 beq 指令的含义，并解释为什么汇编程序在对下列汇编源程序中的 beq 指令进行汇编时会遇到问题，应该如何修改该程序段？

```
here:  beq $s0, $s2, there
        …
there: addi $s1, $s0, 4
```

14. 以下 C 语言程序段中有两个函数 sum_array 和 compare，假定 sum_array 函数先被调用，全局变量 sum 分配在寄存器 \$ s0 中。要求按照 MIPS 过程调用协议写出每个函数对应的 MIPS 汇编语言程序，并画出每个函数调用前、后栈中的状态以及帧指针和栈指针的位置。

```
int sum=0;

int sum_array ( int array[ ], int num )
{
    int i;
    for (i=0; i<num; i++)
         if compare (num, i+1) sum+=array[i] ;
    return sum;
}

int compare ( int a, int b )
{
    if (a>b)
```

```
        return 1;
    else
        return 0;
}
```

15. 以下是一个计算阶乘的 C 语言递归过程，请按照 MIPS 过程调用协议写出该递归过程对应的 MIPS 汇编语言程序，要求目标代码尽量短(提示：乘法运算可用乘法指令“mul rd, rs, rt”来实现，功能为“rd←(rs)×(rt)”)。

```
int fact ( int n)
{
    if (n<1)
        return (1);
    else return (n * fact (n-1) );
}
```

第 6 章 中央处理器

计算机所有功能通过执行程序完成，程序由若干条指令构成。计算机采用“存储程序”的工作方式，也即计算机必须能够自动地从主存取出一条条指令执行，而专门用来执行指令的部件就是中央处理器(Central Processing Unit，CPU)。

在处理器中控制指令执行的部件是控制器，控制器可采用硬连线路方式实现，也可采用微程序设计方式实现，也有一些 CPU 采用硬连线路和微程序控制相结合的方式实现。

本章主要介绍 CPU 的基本功能和基本组成以及单周期处理器和多周期处理器的工作原理和设计方法。有关流水线处理器的基本设计原理在第 7 章介绍。

6.1 CPU 概述

6.1.1 指令执行过程

指令按顺序存放在内存连续单元中，指令地址由 PC 给出。CPU 取出并执行一条指令的时间称为指令周期，不同指令的指令周期可能不同。

图 6.1 给出了如下 CPU 执行指令的过程。

(1) 指令地址计算。顺序执行时，下条指令地址的计算比较简单，只要将 PC 加上当前指令长度即可；当遇到转移等改变执行顺序的指令时，则需要根据指令操作码和寻址方式决定下条指令地址的计算方式。

(2) 取指令。需要一次或多次访存，对于定长指令字格式，通常只要一次访存就能取出一条指令；若指令字长度可变，则可能需要多次访存才能取出指令。

(3) 指令操作码的译码。每条指令的功能不同，所以涉及到的操作过程不同，因而需要不同的控制信号。例如，MIPS 的 R-型 add 指令要求从寄存器取数、做加法及结果送寄存器；而 MIPS 的 I-型 ori 指令则要求从寄存器取数、对立即数进行 0 扩展、“或”运算及结果送寄存器。因而，应该根据指令的不同操作码译出不同的控制信号。

(4) 源操作数地址计算并取操作数。根据寻址方式确定源操作数地址计算方式，若是存储器数据，则需要一次或多次访存；若指令为间接寻址或两个操作数都在存储器的双目运算时，需要多次访存；若是寄存器数据，则直接从寄存器取数后，转到下一步进行数据操作。

(5) 数据操作。在 ALU 或加法器等运算部件中进行运算。

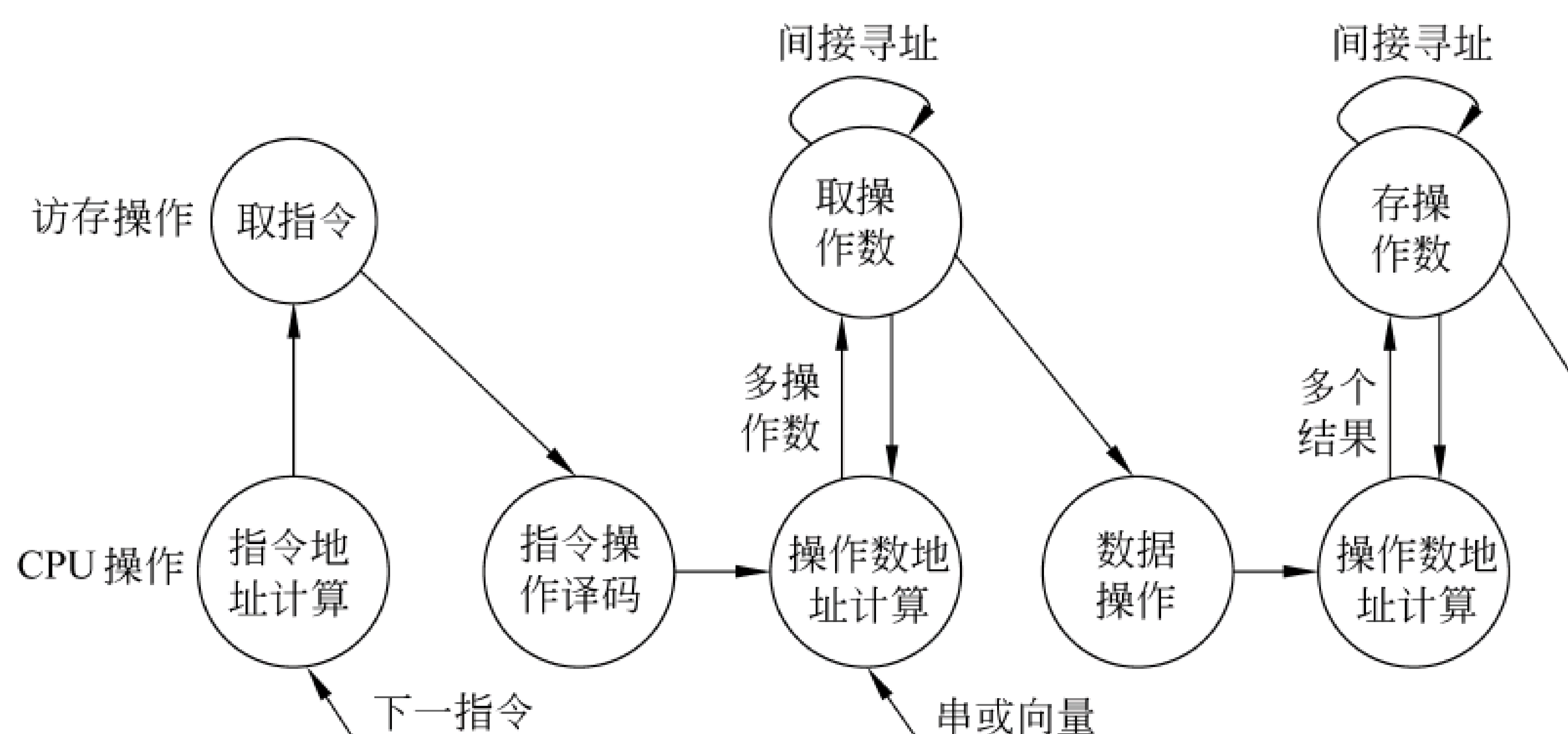


图 6.1　指令执行过程

(6) 目的操作数地址计算并存结果。根据寻址方式确定目的操作数地址计算方式,若是存储器数据,则需要一次或多次访存(间接寻址时);若是寄存器数据,则在进行数据操作时直接存结果到寄存器。

如果是串操作或向量运算指令,则可能会循环执行第(4)～(6)步多次。

通过对上述指令执行过程分析可知,每条指令的功能总是由以下 4 种基本操作来实现。

(1) 读取某个存储单元的内容,并将其装入某个寄存器。

(2) 把一个数据从某个寄存器存入给定的存储单元中。

(3) 把一个数据从某个寄存器送到另一个寄存器或者 ALU。

(4) 进行某种算术运算或逻辑运算,将结果送入某个寄存器。

上述这些基本操作功能可以用形式化的方式来描述,所用的描述语言称为寄存器传送语言 RTL(Register Transfer Language)。本章所用的 RTL 规定如下:

(1) 用 R[r]表示寄存器堆中寄存器 r 的内容。

(2) 用 M[addr]表示读取存储单元 addr 的内容。

(3) 传送方向用"←"表示,传送源在右,传送目的在左。

(4) 程序计数器 PC 直接用 PC 表示其内容。

由此可知,若要表示寄存器间接寻址操作(即读取寄存器 r 的内容所指的存储单元的内容),则用 M[R[r]]表示;若要表示 PC 所指内存单元的内容,则用 M[PC]表示。

6.1.2　CPU 的基本功能

CPU 的基本职能是周而复始地执行指令,但是,在执行指令过程中可能会遇到一些异常情况和外部中断。例如,对指令操作码译码时,发现有不存在的"非法操作码";在访问指令或数据时发现"缺页";外部设备请求中断 CPU 的执行等。所以,CPU 除了执行指令外,还要能够发现和处理"异常"情况和"中断"请求。

具体来说,CPU 的职能有以下几个方面。

(1) 控制指令执行顺序。程序是一个有序的指令序列,指令的执行必须严格按照事先安排的顺序进行,因此,CPU 的一个重要任务是控制好指令的执行顺序。为此,CPU 中必须提供指令地址的保存场所和指令地址的计算部件,必须能够正确地确定每条指令的下条指令地址计算方式。

(2) 控制指令执行操作。每条指令的功能通过执行一系列操作来实现,CPU 必须能够

确定每条指令的操作序列,并通过指令译码生成对这些操作的控制信号,送到执行操作的部件,控制操作正确进行。

(3) 控制操作时序。每条指令的执行都必须有严格的定时控制,一条指令包含的每一步操作之间都有一定的时间顺序,CPU 必须提供对指令操作的定时控制。

(4) 对数据进行运算。CPU 中必须提供实现所有指令功能的运算部件。运算部件的具体实现可参照第 3 章的有关内容。

(5) 对存储器或 I/O 访问进行控制。在指令执行过程中,需要取指令、存取内存数据或访问 I/O 设备,CPU 应能提供对存储器或 I/O 访问的控制,CPU 中用于这部分控制的部件包括总线接口部件(BIU)、存储管理部件(MMU)等。

(6) 异常和中断处理。CPU 必须能够发现和处理"异常"情况及外部"中断"请求。

6.1.3 CPU 的基本组成

随着超大规模集成电路技术的发展,更多的功能逻辑被集成到 CPU 芯片中,包括 cache、MMU、浮点运算逻辑、异常和中断处理逻辑等,因而 CPU 的内部组成越来越复杂,甚至在一个 CPU 芯片中集成了多个处理器核。但是不管 CPU 多复杂,它都可看成由数据通路(Data Path)和控制部件(Control Unit)两大部分组成。

通常将指令执行过程中数据所经过的路径,包括路径上的部件称为数据通路。ALU、通用寄存器、状态寄存器、cache、MMU、浮点运算逻辑、异常和中断处理逻辑等都是指令执行过程中数据流经的部件,都属于数据通路的一部分。通常把数据通路中专门进行数据运算的部件称为执行部件(Execution Unit)或功能部件(Function Unit)。

数据通路由控制部件进行控制。控制部件根据每条指令功能的不同生成对数据通路的控制信号,并正确控制指令的执行流程。

为了在教学上遵循由简到难的原则,首先从 CPU 最基本的组成开始了解。CPU 的基本功能决定了 CPU 的基本组成,图 6.2 所示是 CPU 的基本组成原理图。

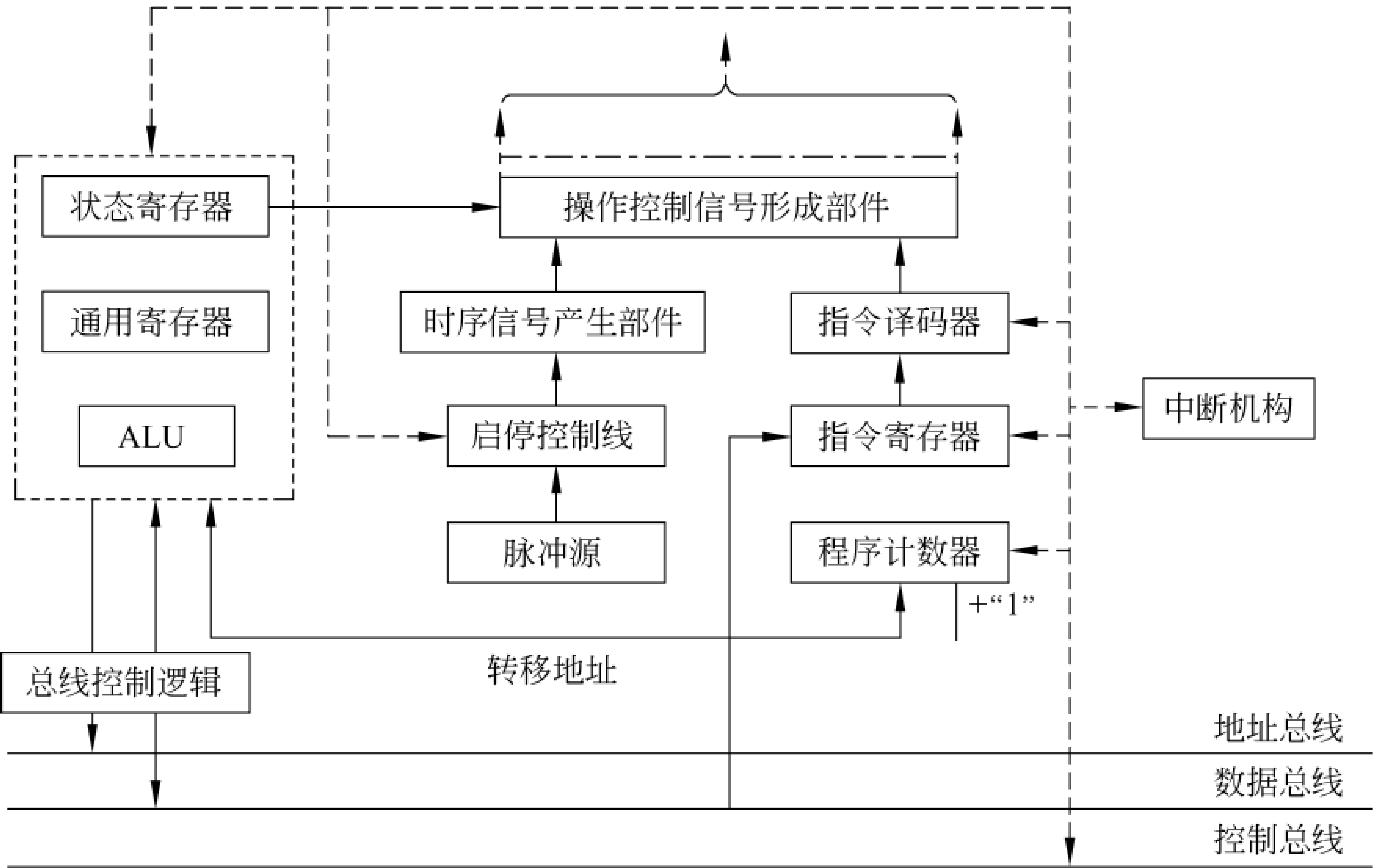


图 6.2　CPU 基本组成原理图

图 6.2 中的数据通路非常简单，只给出了最基本的执行部件，如 ALU、通用寄存器和状态寄存器等。其余部件都是控制逻辑或与其密切相关的逻辑，包括以下几部分。

(1) 程序计数器(PC)。又称指令计数器或指令指针(IP)，用来存放指令的地址。指令地址的形成有两种可能：①顺序执行时，PC+“1”形成下条指令地址。有的机器 PC 本身具有+“1”计数功能，这里的“1”指一条指令的长度；有的机器借用运算部件完成。②需要改变程序执行顺序时，通常由转移类指令形成转移地址送到 PC 中，作为下条指令地址。每个程序开始执行之前，总是把程序中第一条指令的地址送到 PC 中。

(2) 指令寄存器(IR)。用以存放现行指令。上文提到，每条指令总是先从存储器取出后才能在 CPU 中执行，指令取出后存放在指令寄存器中，以便送指令译码器进行译码。

(3) 指令译码器。对指令寄存器中的操作码部分进行分析解释，产生相应的译码信号提供给操作控制信号形成部件。

(4) 脉冲源及启停控制线路。脉冲源产生一定频率的脉冲信号作为整个机器的时钟脉冲，是 CPU 时序的基准信号。启停线路在需要时能保证可靠地开放或封锁时钟脉冲、控制时序信号的发生与停止并实现对机器的启动与停机。

(5) 时序信号产生部件。以时钟脉冲为基础，产生不同指令对应的周期、节拍、工作脉冲等时序信号，实现机器指令执行过程的时序控制。

(6) 操作控制信号形成部件。综合时序信号、指令译码信号和执行部件反馈的状态标志等，形成不同指令所需要的操作控制信号序列。

(7) 总线控制逻辑。实现对总线传输的控制，包括数据、地址信息的缓冲与三态控制。

(8) 中断机构。实现对异常情况和某些外部中断请求的处理。

6.1.4 数据通路的基本结构

指令执行所用到的元件有两类：组合逻辑元件(也称操作元件)和存储元件(也称状态元件)。连接这些元件的方式有两种：总线方式和分散连接方式。所以，数据通路就是由操作元件和存储元件通过总线或分散方式连接而成的进行数据存储、处理和传送的路径。

1. 组合逻辑(操作)元件

组合逻辑元件的特点是，输出只取决于当前的输入。即若输入一样，其输出也一样。组合电路的定时不受时钟信号的控制，所有输入信号到达后，经过一定的逻辑门延迟，输出端的值被改变，并一直保持其值不变，直到输入信号改变。

数据通路中常用的组合逻辑元件有多路选择器 MUX、加法器 Adder、算术逻辑部件 ALU、译码器 Decoder 等，其符号表示如图 6.3 所示。

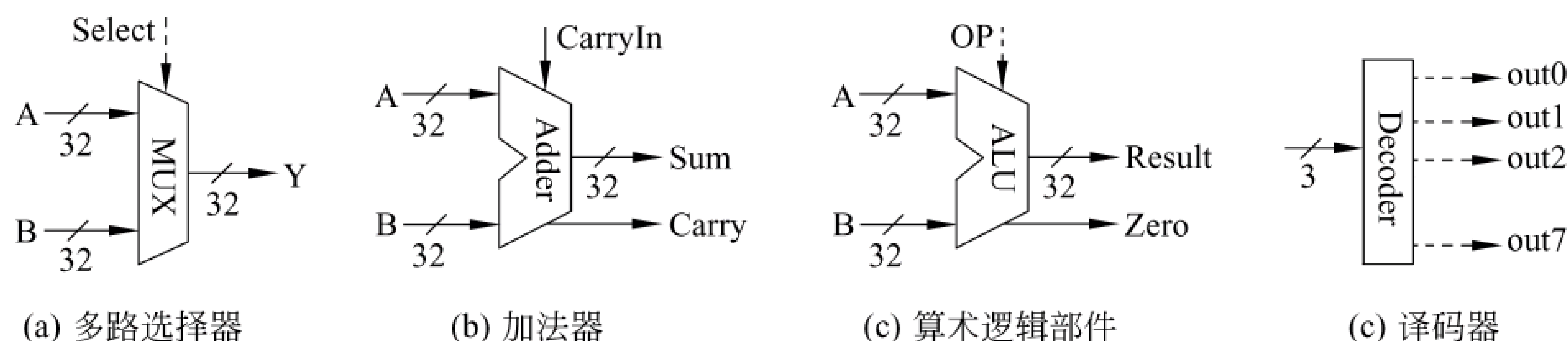


图 6.3 数据通路中的常用组合逻辑元件

图中虚线表示控制信号，多路选择器需要控制信号Select确定选择哪个输入被输出；加法器不需要控制信号控制，因为它的操作是确定的；ALU需要有操作控制信号op，由它确定ALU进行哪种操作；译码器通过对指令操作码进行译码，输出译码信号out0、out1、…，译码器无须控制信号进行控制。

2. 状态（存储）元件

状态（存储）元件的特点是，具有存储功能，输入状态在时钟控制下被写到电路中，并保持电路的输出值不变，直到下一个时钟到达。输入端状态由时钟决定何时被写入，输出端状态随时可以读出。最简单的状态单元是D触发器，有一个时钟输入、一个状态输入和一个状态输出，图6.4是D触发器的定时示意图。

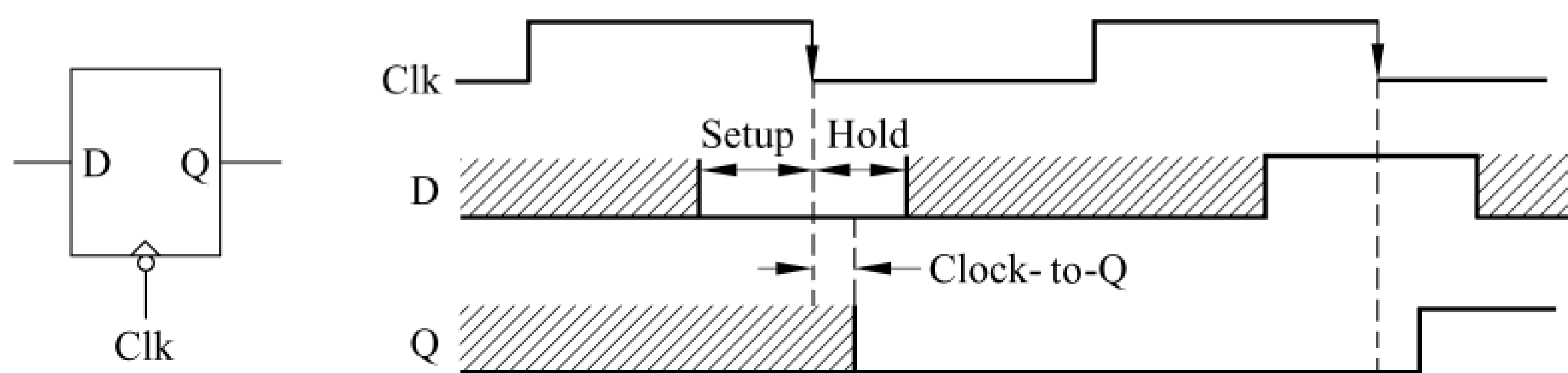


图6.4　D触发器定时示意图

图中D触发器采用时钟下降沿触发，要使输出状态能正确地随着输入状态改变，必须满足以下时间约束：(1)在时钟下降沿到来前的一定时间内，输入端D的状态必须稳定有效，这段时间被称为建立时间(Setup Time)；(2)在时钟下降沿到来后的一定时间内，输入端D的状态必须继续保持稳定不变，这段时间被称为保持时间(Hold Time)。在上述两个约束条件满足的情况下，经过时钟下降沿到来后的一段延迟时间("Clock-to-Q"time)，输出端Q的状态改变为输入端D的状态，并一直保持不变，直到下个时钟到来。

数据通路中的寄存器是一种典型的状态存储元件，由 n 个D触发器可构成一个 n 位寄存器。根据功能和实现方式的不同，有各种不同类型的寄存器。例如，(1)带"写使能"输入信号的触发器构成的寄存器：通常称这类触发器为锁存器(Latch)，所组成的寄存器称为暂存器，通常用来实现数据通路中的指令寄存器IR、通用寄存器组(General Register Set)①等；(2)输出端带一个三态门的寄存器：通常用于与总线相连的寄存器，可通过三态门控制信息是否打到总线上；(3)带复位(清0)功能的寄存器；(4)带计数(自增)功能的寄存器；(5)带移位功能的寄存器。这些不同类型的寄存器都在时钟信号和相应控制信号(如"写使能"、"三态门开"、"清0"、"自增"和"左移/右移")的控制下完成信息存储功能。当然，也可以将上述功能组合起来构成寄存器。图6.5是数据通路中的暂存寄存器和通用寄存器组的外部结构示意图。

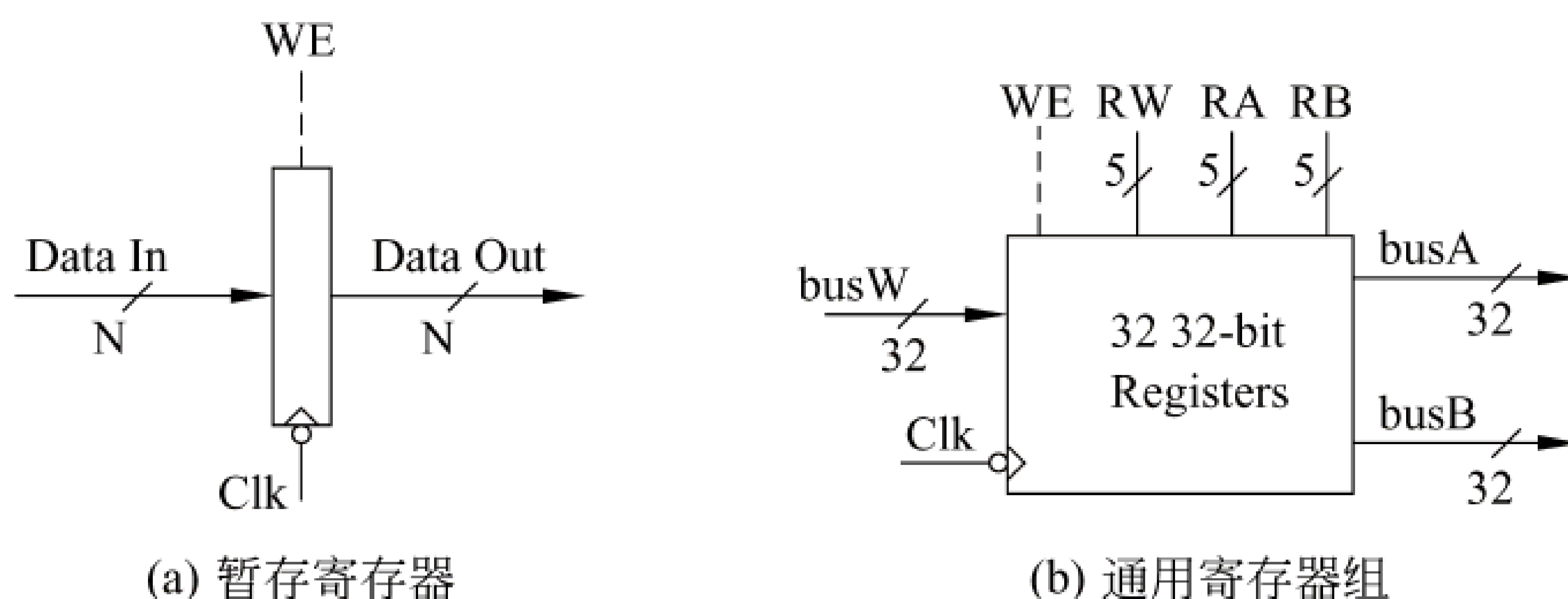
(1) 暂存寄存器

有一个写使能(Write Enable)信号WE，其功能定义如下。

WE=0：时钟信号(Clk)边沿到来时，不会改变输出值。

WE=1：时钟边沿到来后，经过Clk-to-Q时间的延迟，输出端开始变为输入端的值。

① 通用寄存器组(General Register Set)：简称GRS，有的英文原版教材用Register Files表示，翻译为寄存器堆。



(a) 暂存寄存器　　(b) 通用寄存器组

图 6.5　寄存器和寄存器组的外部结构

若数据通路中某个寄存器在每个时钟到来时都要写入信息，则可以不要 WE 信号。

(2) 通用寄存器组

图 6.5(b)所示的是一个带时钟控制的通用寄存器组，有两个读口，busA 和 busB 分别由 RA 和 RB 给出地址。读操作属于组合逻辑操作，无须时钟控制。即当地址 RA 或 RB 有效后，经过一个“取数时间”的延迟，在 busA 和 busB 上的信息有效。有一个写口，busW 上的信息写入的地址由 RW 指定。写操作属于时序逻辑操作，需要时钟信号的控制。在 WE 为 1 的情况下，时钟边沿到来后经过 Clk-to-Q 时间延迟，busW 传来的值开始被写入 RW 指定的寄存器中。

3. 数据通路与时序控制

指令执行过程中的每个操作步骤都有先后顺序，为了使计算机能正确执行指令，CPU 必须按正确的时序产生操作控制信号。由于不同指令对应的操作序列长短不一，序列中各操作执行时间也不相同，因此，需要考虑用怎样的时序方式来控制。

(1) 早期计算机的三级时序系统

早期计算机通常采用机器周期、节拍和脉冲三级时序对数据通路操作进行定时控制。一个指令周期可分为取指令、读操作数、执行并写结果等多个基本工作周期，称为机器周期。有取指令、存储器读、存储器写、中断响应等类型的机器周期。

每个机器周期长短可能不同，例如，存储器读或写周期比 CPU 中的操作时间长得多。所以，机器周期的宽度通常以主存工作周期为基础来确定。

一个机器周期内要进行若干步动作。例如，存储器读周期有送地址、发读命令、检测数据有无准备好、取数据等。因此，有必要将一个机器周期再划分成若干节拍，每个动作在一个节拍内完成。

为了产生操作控制信号并使某些操作能在一个节拍时间内配合工作，常在一个节拍内再设置一个或多个工作脉冲。例如，要在一个节拍内使某个寄存器的内容送到另一个寄存器中，就需要设置先后两个工作脉冲，以产生打开数据通路脉冲和接受脉冲。图 6.6 是机器周期、节拍和脉冲三级时序系统示意图。图中假定每个机器周期有 4 个节拍，每个节拍有 4 个脉冲。

(2) 现代计算机的时钟信号

现代计算机中，已不再采用上述三级时序系统，机器周期的概念已逐渐消失。整个数据通路中的定时信号就是时钟，一个时钟周期就是一个节拍。

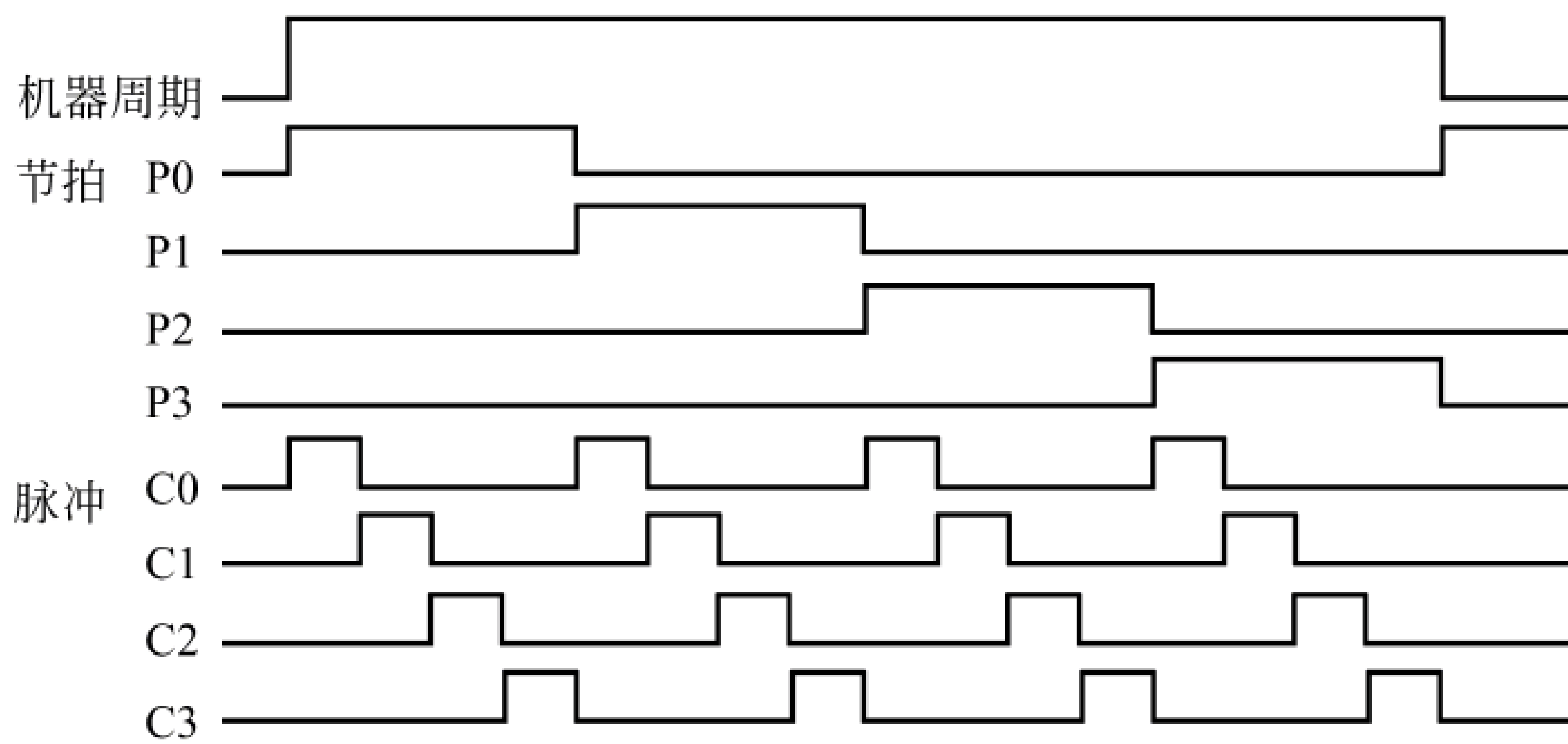


图 6.6 机器周期、节拍、脉冲三级时序系统

如图 6.7 所示，数据通路可看成由组合逻辑（操作）元件和状态（存储）单元交替组合而成，即数据通路的基本结构为“……—状态单元—操作元件（ 组合电路）—状态单元—……”。

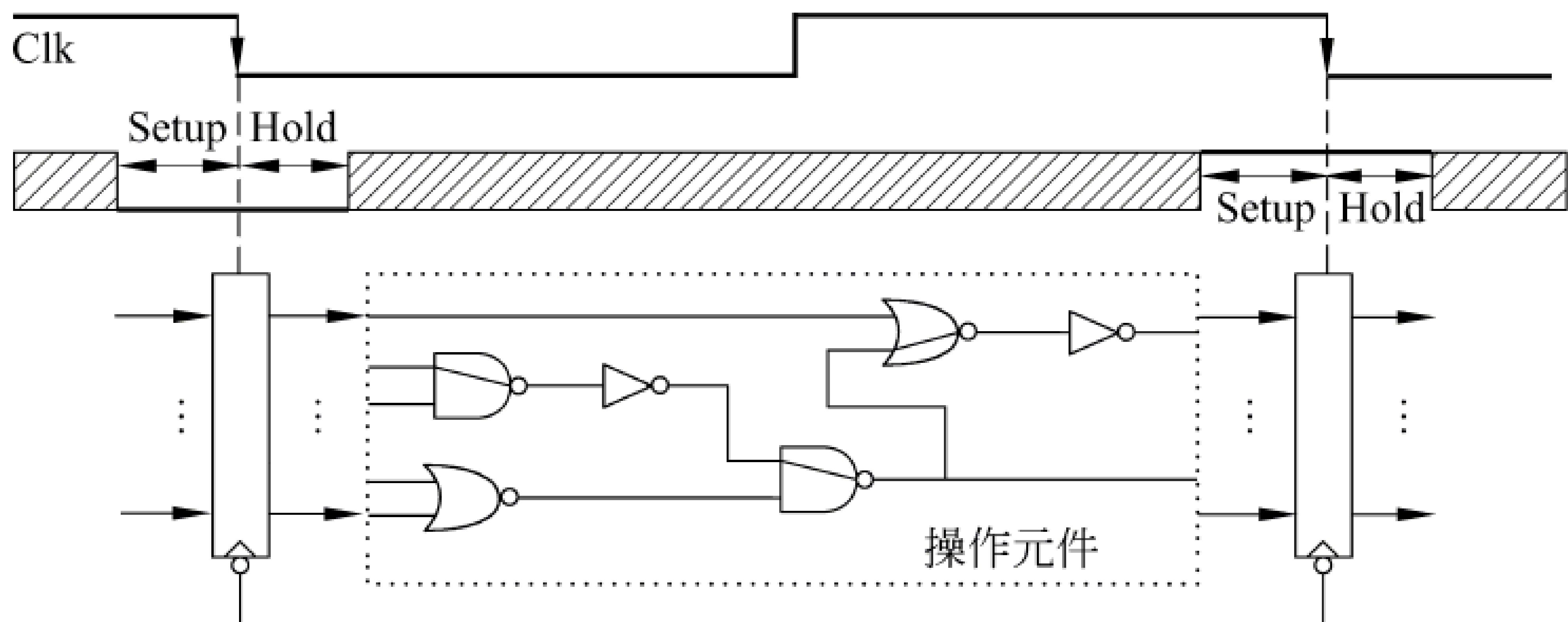


图 6.7 数据通路和时钟周期

只有状态元件能存储信息，所有操作元件都须从状态单元接收输入，并将输出写入状态单元中。所有状态单元在同一时钟控制下写入信息。假定采用下降沿（负跳变）触发方式，则所有状态单元在时钟下降沿到来时开始写入信息，经过触发器的锁存延迟（Latch Prop，即 Clk-to-Q 时间）后输出开始有效。假定每个时钟的下降沿是一个时钟周期的开始时刻，则一个时钟周期内整个处理过程如下：经过 Clk-to-Q 时间，前一个时钟周期内生成的信号被写入状态单元，并输出到随后的操作元件进行处理，经过若干级门延迟，得到的处理结果被送到下一级状态单元的输入端，然后必须稳定一段时间（Setup Time）才能开始下个时钟周期，并在时钟信号到达后还要保持一段时间（Hold Time）。

假定所有各级操作元件中最长操作延迟时间为 Longest Delay，考虑时钟偏移（Clock Skew）[①]，根据上述分析可知，数据通路的时钟周期 Cycle Time 应为 Cycle Time＝Latch Prop＋Longest Delay＋Setup Time＋Clock Skew。假定各级操作元件中最短操作延迟时间为 Shortest Delay，为了数据通路正常工作，应该满足以下时间约束：Latch Prop＋

① 时钟偏移（Clock Skew）：由于器件工艺和走线延迟等原因造成的同步系统中时钟信号的偏差，这种时间偏差使得时钟信号不能同时到达不同的状态元件而导致同步定时错误，所以需要在时钟周期中增加时钟偏移时间来避免这种错误。也有人把 Clock Skew 翻译为时钟扭斜。

Shortest Delay>Hold Time。

*** 4. 早期累加器型指令系统数据通路**

1946 年冯·诺依曼和他的同事在普林斯顿高级研究院开始设计存储程序计算机，它被称为 IAS(普林斯顿高级研究院 Institute for Advanced Study 的简称)计算机，是后来通用计算机的原型。IAS 所使用的数据通路中，存储部件除了主存 M 外，有累加器 AC、乘商寄存器 MQ、指令寄存器 IR、程序计数器 PC，另外，还有主存缓冲寄存器 MBR 和主存地址寄存器 MAR，CPU 和外部的数据和地址交换都通过这两个寄存器进行。图中的 IBR 是指令缓冲寄存器，可看成 IR 的一部分。图 6.8 所示的就是冯·诺依曼结构机器最早使用的累加器型数据通路，也是最简单的数据通路结构。

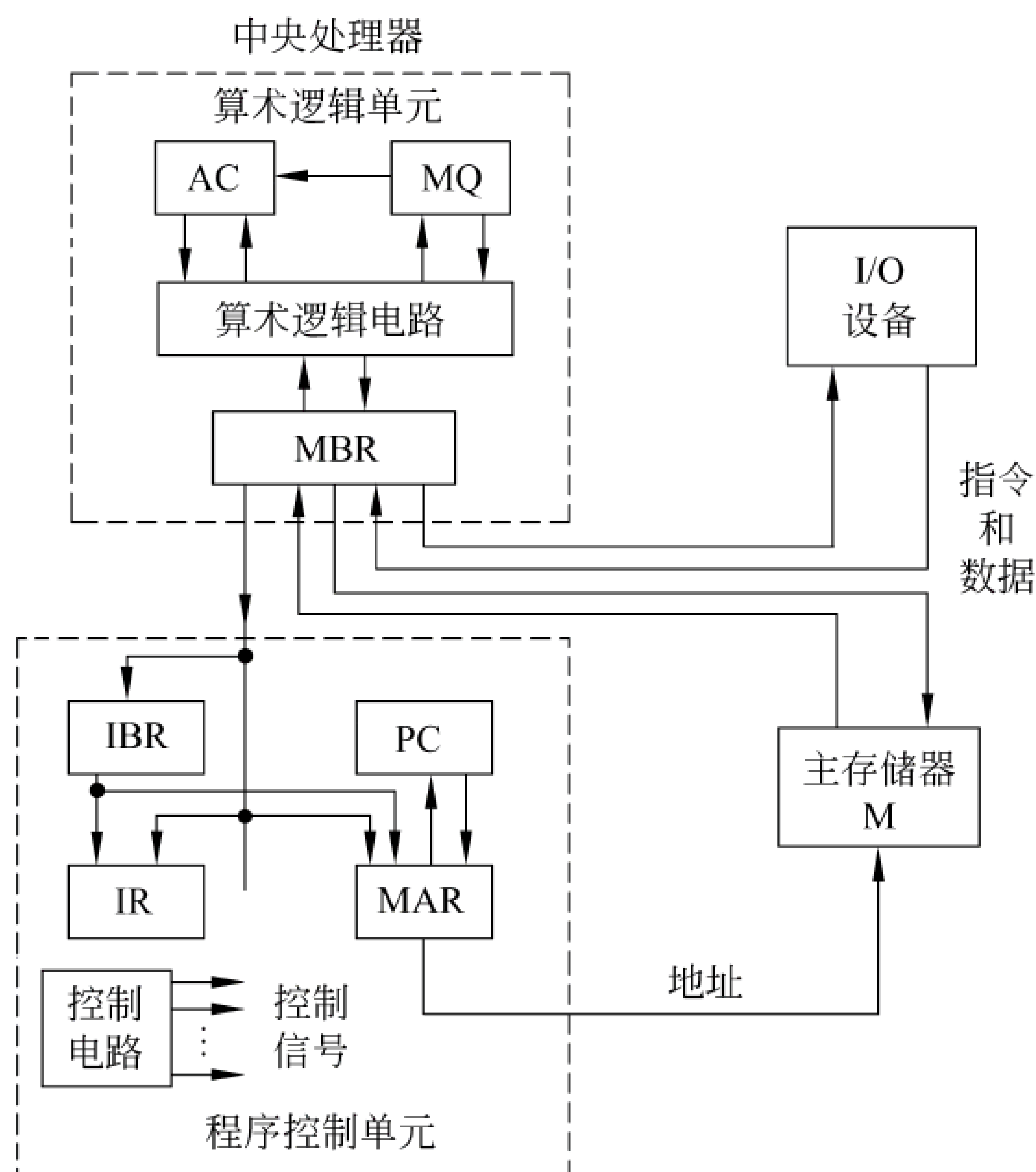


图 6.8　IAS 计算机数据通路

IAS 计算机数据通路是累加器型指令系统的典型结构。算术逻辑电路(ALU)的一个操作数总是来自累加器 AC，另一个操作数来自主存，通过 MBR 送到 ALU 的输入端，ALU 运算后的中间结果送到 AC 或乘商寄存器 MQ，通过对 ALU 的控制，可以实现加、减、乘、除和与、或、非等各种算术、逻辑运算。取指令的数据路径为：PC→MAR，Read M，M→MBR→IBR→IR；取操作数、运算、送结果的数据路径为：操作数地址→MAR，Read M，M→MBR→ALU 输入端，AC→ALU 输入端，ALU 操作，ALU 结果→MBR，Write M。

5. 单总线数据通路

CPU 内部的组合逻辑电路和存储部件也可通过总线方式连接。如图 6.9 所示，将 ALU 及所有寄存器通过一条内部的公共总线连接起来，构成单总线结构的数据通路。因为此总线在 CPU 内部，所以称为 CPU 内部总线，不要把它与连接 CPU、存储器和 I/O 设备的外部系统总线(如图中的存储器总线)相混淆。

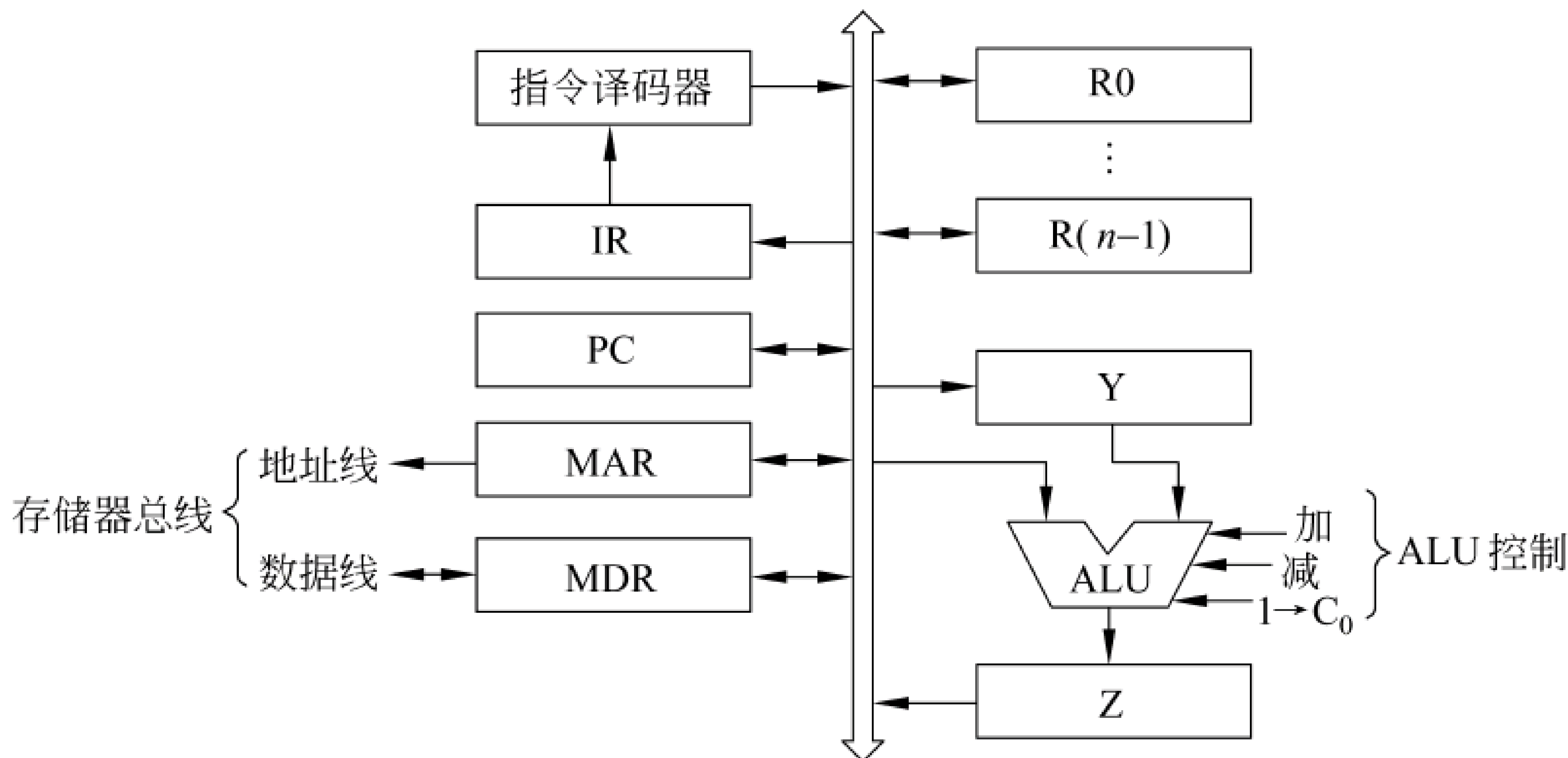


图 6.9　单总线数据通路

图 6.9 中，寄存器 R0 到 R(n－1)是程序员可见的通用寄存器，而寄存器 Y 和 Z 对程序员而言是透明的，仅被 CPU 用作某些指令执行期间的临时存储单元。指令寄存器 IR 和指令译码器是 CPU 内部控制单元的主要部件。存储器总线经由存储器数据寄存器 MDR 和存储器地址寄存器 MAR 连到 CPU。为了简化以下 4 种基本操作的说明，假定 MAR、MDR 与存储器总线之间没有三态门控制，它们的输出一直处于使能状态。

在图 6.9 的单总线 CPU 结构中，完成指令执行的 4 种基本操作(参见 6.1.1 节)过程说明如下。

(1) 在寄存器之间传送数据

总线是一组共享的传输信号线，它不能存储信息，某一时刻也只能有一个部件能把信息送到总线上。在图 6.9 的单总线 CPU 结构中，连到内部总线上的各个部件之间通过内总线传送数据。源寄存器将信息送到总线上，经过总线上一定的延迟，信息被传送到目的寄存器并被存储在目的寄存器中。通常在寄存器和总线之间有两个控制信号：R_{in} 和 R_{out}，R_{in}＝1 时控制将总线上的信息存到寄存器 R 中；R_{out}＝1 时，控制寄存器 R 将信息送到总线。

图 6.10 给出了寄存器中的一位触发器和内总线相连时的控制电路和控制信号，对于一个由 n 个触发器构成的 n 位寄存器，其原理是一样的。在图 6.10 中，D 触发器的数据输入端连到一个二路多选器，当控制信号 R_{in}＝1 时，选择总线上的信息输入到 D 触发器的输入端，当时钟信号的上升沿到达时，被装入到触发器中；当 R_{in}＝0 时，触发器的值不变。D 触发器的输出端通过一个三态门与总线相连，当控制信号 R_{out}＝1 时，三态门被打开，触发器的输出被送到总线上；当 R_{out}＝0 时，则三态门的输出端呈高阻态，触发器与总线断开。

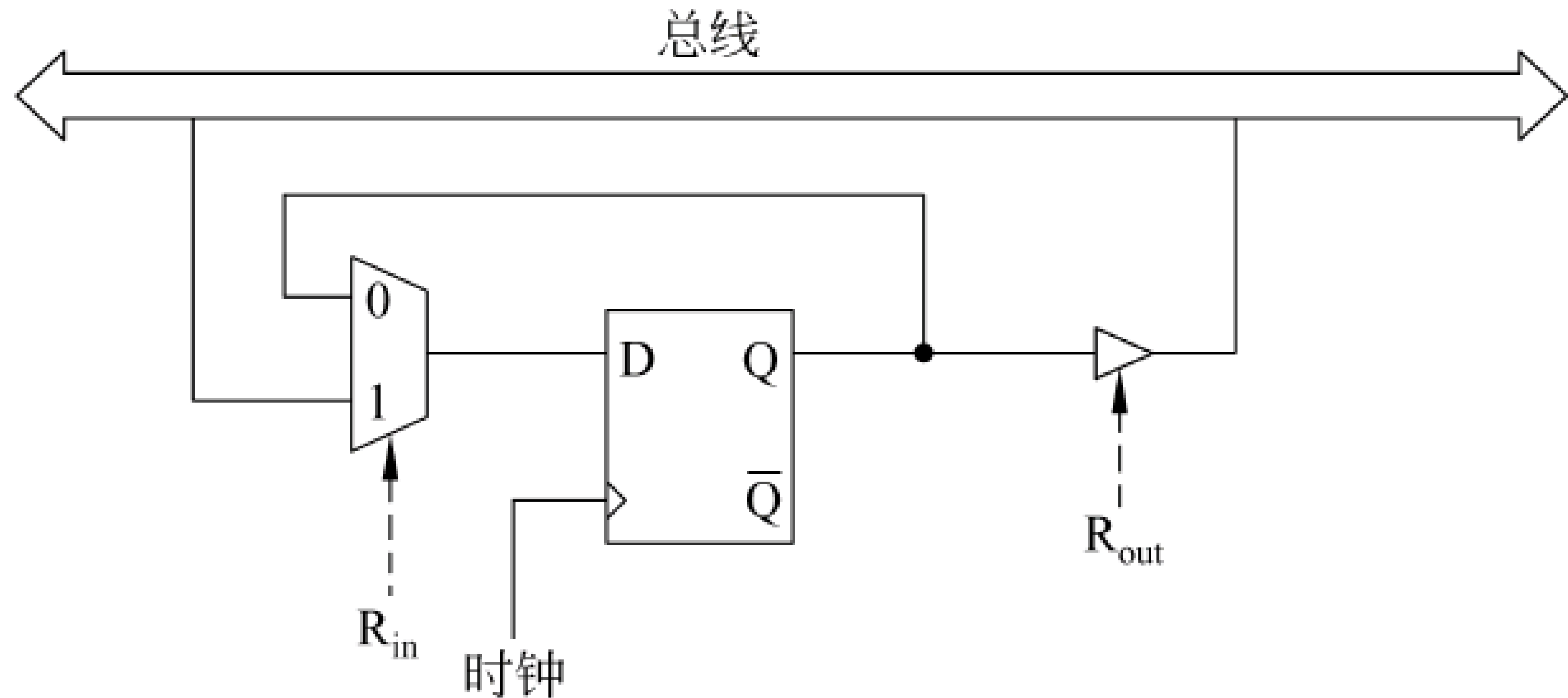


图 6.10　一位寄存器的输入输出控制

在图 6.9 所示的数据通路中，要将寄存器 R0 的内容传送到寄存器 Y，则对应的控制信号为 $R0_{out}$，Y_{in}。

这里必须说明，在很多计算机的 CPU 内部结构中，寄存器堆之间没有直接的通路，因此寄存器之间的数据传送需经过 ALU 来实现，此时，控制操作要相应复杂一些。

（2）完成算术、逻辑运算

ALU 是一个没有记忆功能的组合逻辑电路，若要进行正确的运算，必须将两个操作数都送到 ALU 的输入端。在图 6.9 所示的数据通路中，Y 寄存器存放其中一个操作数，另一个操作数被置于总线上。运算结果被临时存放在寄存器 Z 中。因此，要实现操作：R[R3]←R[R1]＋R[R2]，即寄存器 R1 的内容与 R2 的内容相加，结果送寄存器 R3，则控制信号如下。

① $R1_{out}$，Y_{in}。

② $R2_{out}$，add，Z_{in}。

③ Z_{out}，$R3_{in}$。

以上三步不能同时执行，因为任何时刻只能有一个寄存器的输出送到总线上。因此，该操作需要三个时钟周期（节拍）。

在上述第②步加法操作中，信号 $R2_{out}$ 被置 1 后，输出三态门被接通，数据沿总线传输到 ALU 的输入端，并在 add 信号的控制下，在 ALU 中做加法运算，最后在 Z_{in} 的控制下，将结果写入 Z。为了能正确实现这一步操作，$R2_{out}$、add 和 Z_{in} 信号必须保持一定的有效时间。

如图 6.11 所示，由于指令译码线路等控制逻辑的延迟，控制信号总是在时钟信号开始一段时间之后才有效，把这段时间记为 Clk-to-Signal，它的时延一定大于锁存延迟 Clk-to-Q，因此，在控制信号开始有效的 t_0 时刻，寄存器 R2 的输出已经有效，此时，$R2_{out}$ 开始作用于三态门，到达 t_1 时刻三态门打开，R2 的内容被送到总线上，经过总线传输和 ALU 延迟，在 t_3 时刻运算结果到达寄存器 Z 的输入端，再经过一段建立时间，在 t_4 时刻稳定，下个时钟到来后就开始写寄存器 Z，为了正确写入 Z，运算结果还必须在 Z 的输入端继续稳定一段保持时间。因此，为了保证能正确完成 ALU 运算并将运算结果正确写入 Z 寄存器，$R2_{out}$ 必须至少保持三态门接通时间、总线传输时间、ALU 延迟、寄存器 Z 的建立和保持时间之和，即 $R2_{out}$ 必须持续保持到 t_5 时刻，同样，add 信号也必须持续保持到 t_5 时刻。

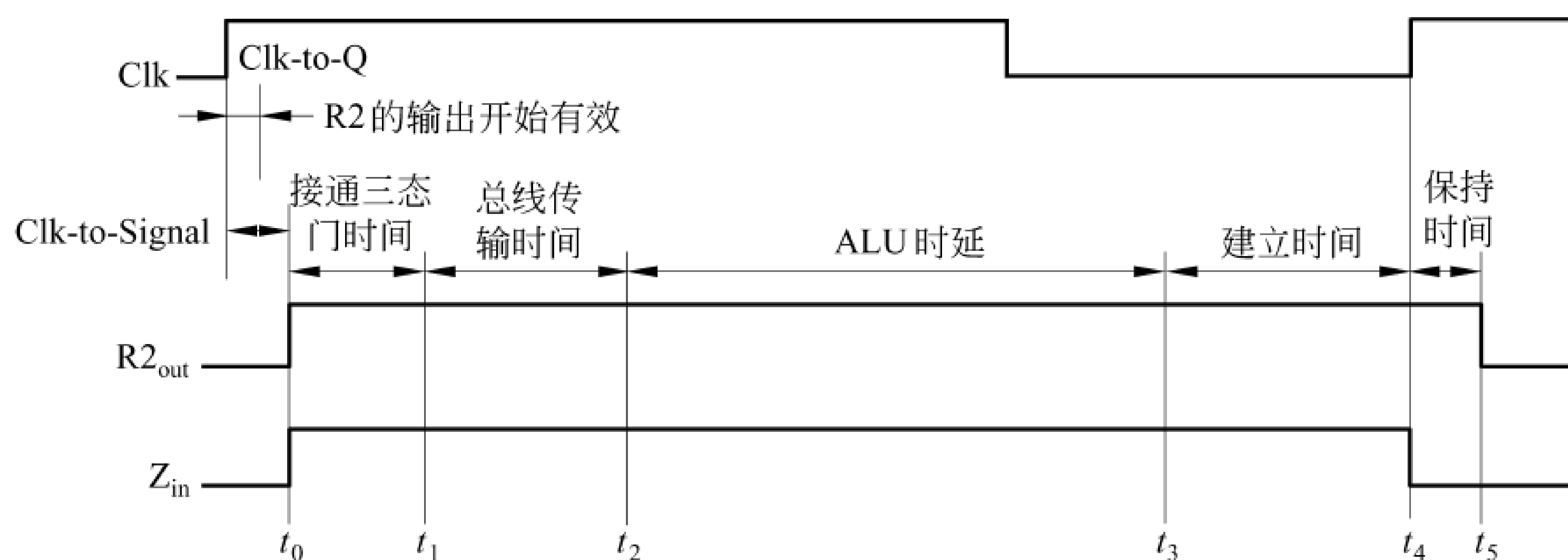


图 6.11　ALU 操作控制信号的定时

（3）从内存读取一个字（指令或数据）

通常，在 CPU 发出地址信息和读信号之后，CPU 必须等待直到主存完成读操作才能继

续执行指令。CPU 如何知道主存是否完成所请求的读操作呢？通常，CPU 和主存之间有"同步"和"异步"两种通信方式。

"异步"方式下，CPU 送出一个读信号(read)来启动一次主存读操作，然后等待主存发回"操作完成"信号。存储器一旦完成读操作，即向 CPU 发回一个存储器功能完成信号 MFC(Memory-Function-Completed)，CPU 在等待期间，在每个时钟到来时采样 MFC 信号，若检测到 MFC 信号有效，则表明存储器已将数据读出，因为前面已经假定 MDR 的外部输入一直处于使能状态，因而存储总线上的数据被直接写入 MDR 中，此时，CPU 可直接从 MDR 取回数据，并继续执行指令。为了使 CPU 能从执行状态转入等待状态，需要有一个控制信号，这里用 WMFC(Wait MFC)表示该控制信号。

假定某条指令中要实现操作：R[R2]←M[R[R1]]，该操作的含义为：将寄存器 R1 所指的主存单元内容装入寄存器 R2，则在图 6.9 所示的数据通路中完成该操作的控制信号序列如下。

① $R1_{out}$，MAR_{in}。

② read，WMFC。

③ MDR_{out}，$R2_{in}$。

第①步通过内部总线将 R1 的内容送到 MAR 的输入端，操作在一个时钟周期内完成。在下个时钟到来后经过一个锁存延迟 Clk-to-Q，MAR 输入端的地址被送到地址总线；第②步在 CPU 发出 read 命令的同时，使 CPU 转入等待状态，处于等待状态的时间取决于所用存储器的速度。通常，从主存读取一个字的时间比在 CPU 内部进行一个操作所花费的时间要长得多，需要多个时钟周期。因此，这一步不能在一个时钟周期内完成；第③步当 CPU 采样到 MFC 信号有效后，就直接将 MDR 中的内容通过内总线送到 R2，这一步操作在一个时钟周期内完成。

"同步"方式下，存储器总是在读信号发出后的固定时钟周期内准备好数据，因而 CPU 不必等待主存发回 MFC 信号，也即"同步"方式下第②步不需要 WMFC 信号。目前，由于主存基本上采用 SDRAM 芯片(参见第 4 章 4.2.3 节)，所以主存与 CPU 之间大多采用"同步"方式进行通信。

(4) 把一个字(数据)写入主存

操作 M[R[R2]]←R[R1]的含义是：将寄存器 R1 的内容写入寄存器 R2 所指的主存单元中。在图 6.9 所示的数据通路中完成该操作的控制信号序列如下。

① $R1_{out}$，MDR_{in}。

② $R2_{out}$，MAR_{in}。

③ write，WMFC。

第③步和上述读内存操作的第②步一样，通常要有多个时钟周期。此外，若主存采用像 SDRAM 芯片这样的同步 DRAM 芯片，则不需要 WMFC 信号。

如果第①步和第②步两个传送不使用相同的物理通道，例如，不送到同一个总线上，则可同时执行。显然，在图 6.9 所示的单总线结构中，这是不允许的。

***6. 三总线数据通路**

要使机器性能提高，必须使每条指令的时钟周期数尽量少。单总线 CPU 中一个时钟

周期内只允许传一个数据，因而其指令执行效率很低。因此，有些CPU内部采用多总线结构。

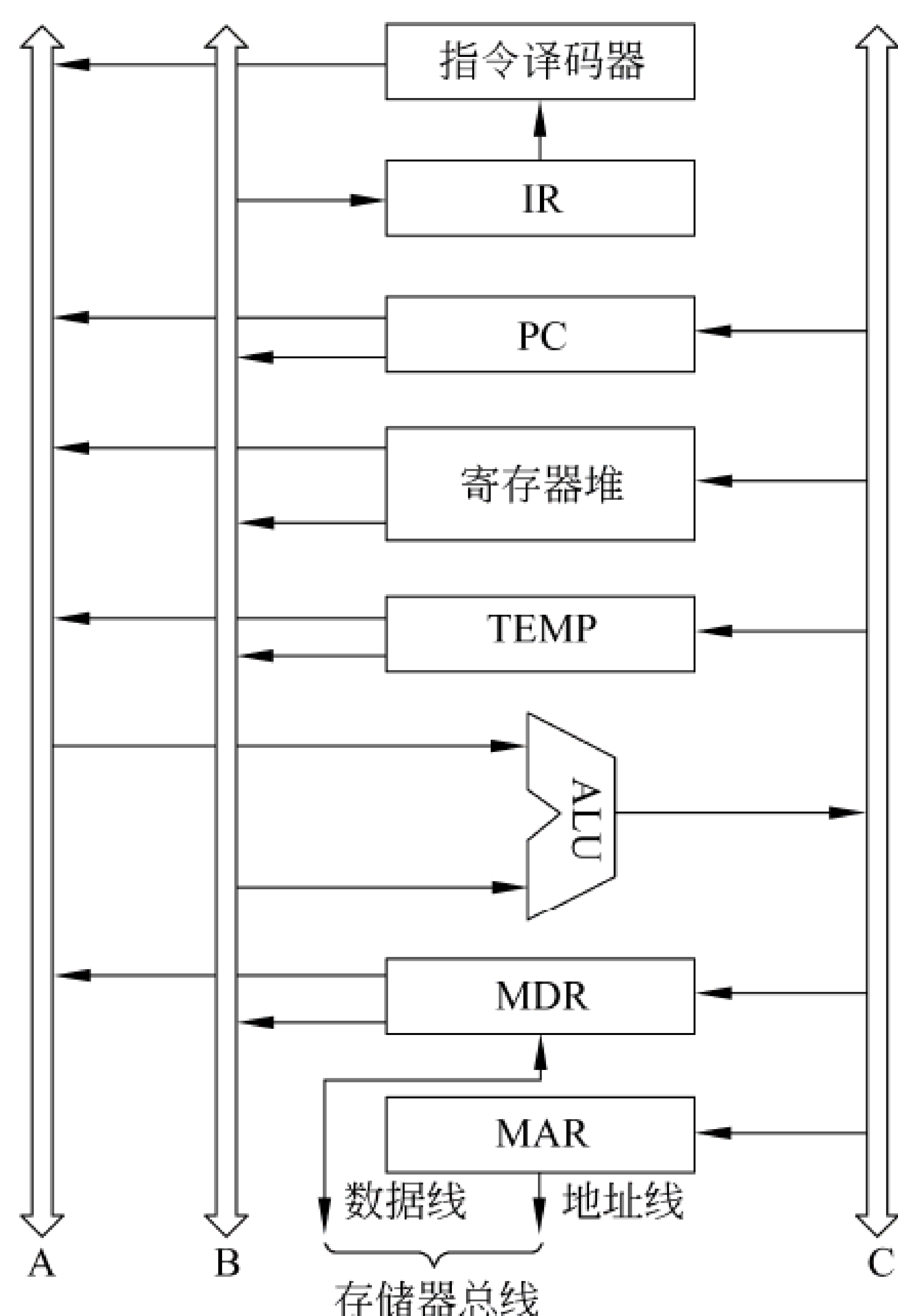


图6.12　三总线数据通路

图6.12给出了三总线结构数据通路示意图。所有通用寄存器在一个双口寄存器堆(register file)中。允许两个寄存器的内容同时输出到A总线和B总线。

与单总线结构相比，多总线结构在执行指令时所需要的步骤大为减少。例如，假定三操作数指令“op R1,R2,R3”的功能为：R[R3]←R[R1] op R[R2]，则可用总线A和B传送源操作数，总线C传送目的操作数。源总线和目的总线之间的连接通过ALU实现。假定所需操作通过ALU一次就可完成，那么，该指令可在一个时钟周期内完成，所需的控制信号为：$R1_{outA}$，$R2_{outB}$，op，$R3_{inC}$。

该数据通路中，若将一个寄存器内容传送到另一个寄存器，则需要通过ALU来完成，实际上只要控制ALU进行“直送”操作即可。三总线结构中，临时寄存器Y和Z都可以不要，这是因为ALU的输入通路分别是总线A和总线B，输出通路为总线C，三者无冲突，而在图6.9的单总线数据通路中，如果缺少了Y或Z，则ALU的输入操作数和输出结果中必定有两个数据同时被送到同一个总线上，因而会发生总线数据冲突。

目前，几乎所有处理器都采用流水线方式执行指令，而采用上述单总线或三总线方式连接的数据通路很难实现指令流水执行。为了更好地理解处理器设计技术，本章后面主要介绍单周期处理器设计和多周期处理器设计，流水线处理器设计技术在第7章介绍。

6.2　单周期处理器设计

处理器设计涉及到数据通路的设计和控制逻辑的设计，其设计过程如下。

第一步：分析每条指令的功能。

第二步：根据指令的功能给出所需的元件，并考虑如何将它们互连。

第三步：确定每个元件所需控制信号的取值。

第四步：汇总各指令涉及的控制信号，生成反映指令与控制信号之间的关系表。

第五步：根据关系表，得到每个控制信号的逻辑表达式，据此设计控制电路。

一个指令系统常常有几十到几百条指令，实现一个完整指令系统的处理器是一项非常复杂、繁琐的任务。为了能清楚说明处理器设计过程与方法，本节以实际的MIPS指令系统为例来说明具体的设计过程。有关MIPS指令系统参见第5章5.4.1节，图6.13再次给出

了 MIPS 的三种指令格式。由于篇幅的限制，不可能介绍所有指令的实现，为此，选择了具有代表性的若干条指令作为实现目标。

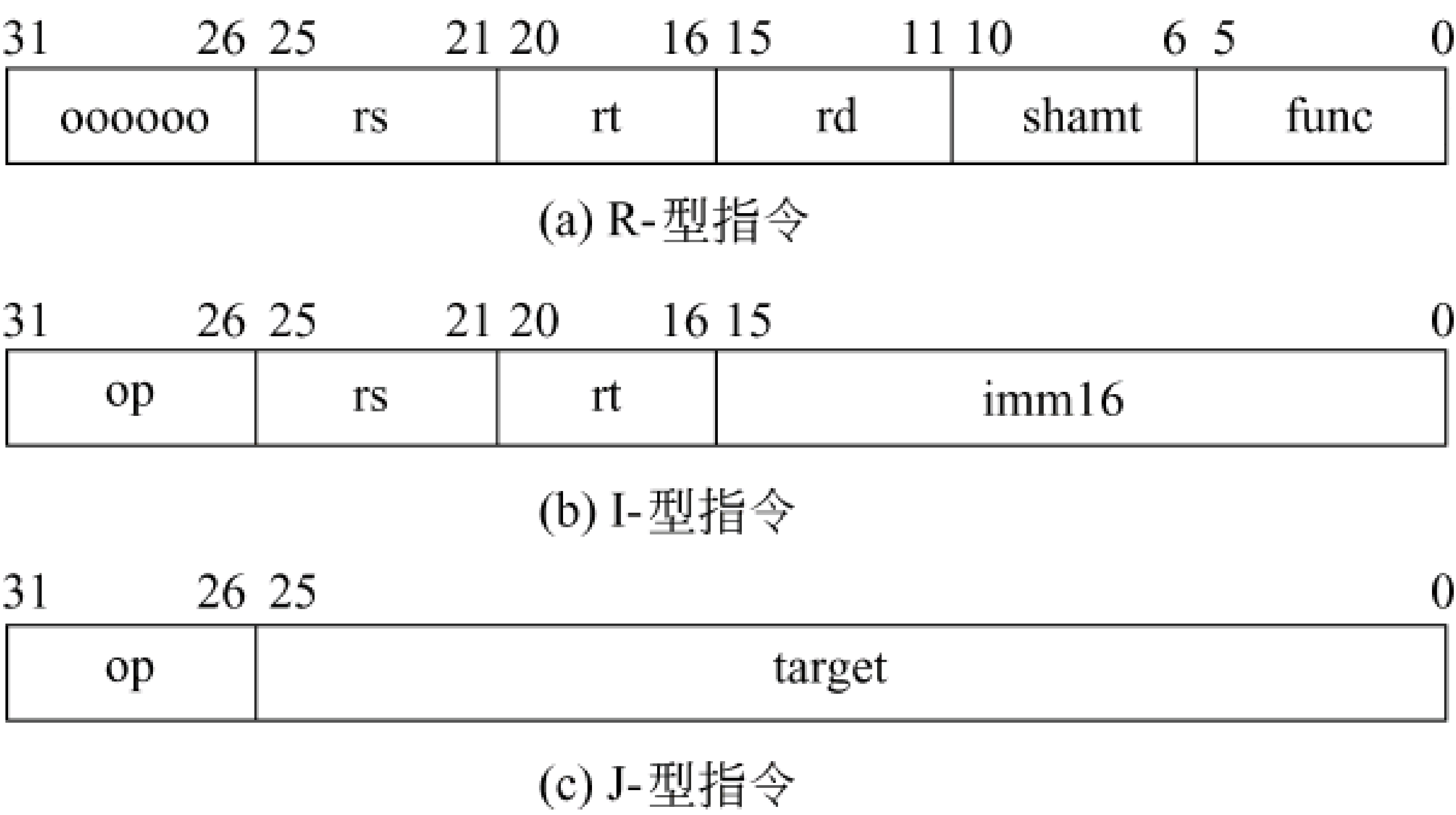


图 6.13　MIPS 指令格式

本节选择以下 11 条 MIPS 指令作为实现目标。

5 条 R-型指令：

```
add  rd, rs, rt
sub  rd, rs, rt
subu rd, rs, rt
slt  rd, rs, rt
sltu rd, rs, rt
```

5 条 I-型指令：

```
ori    rt, rs, imm16
addiu  rt, rs, imm16
lw     rt, rs, imm16
sw     rt, rs, imm16
beq    rs, rt, imm16
```

1 条 J-型指令：

```
j  target
```

这些指令比较具有代表性，包含了 R-型、I-型和 J-型三种类型指令；既有算术/逻辑运算指令，又有取数/存数指令；既有条件转移指令，又有无条件转移指令；既有需要考虑溢出判断的指令，又有无须考虑溢出的指令；既有对带符号数判断大小的指令，又有对无符号数判断大小的指令。关于这些指令的介绍内容，涵盖了大部分指令的基本实现技术。

6.2.1　指令功能的描述

设计处理器的第一步先要确认每条指令的功能，表 6.1 给出了上述 11 条 MIPS 指令的 RTL 描述。其 RTL 描述采用 6.1.1 节的规定。因为每条指令的第一步都是取指令并 PC 加 4 使 PC 指向下条指令，所以表中除第一条 add 指令外，其余指令都省略了对第一步的描述。

在对所有指令进行功能分析的基础上，可以进行数据通路的设计。

表 6.1　11 条目标指令功能的 RTL 描述

指　令	功　能	说　　明
add rd, rs, rt sub rd, rs, rt	M[PC], PC←PC+4 R[rd]←R[rs]±R[rt]	从 PC 所指的内存单元中取指令，并 PC 加 4 从 rs、rt 中取数后相加减，若溢出则异常处理，否则结果送 rd
subu rd, rs, rt	R[rd]←R[rs]－R[rt]	从 rs、rt 中取数后相减，结果送 rd(不进行溢出判断)
slt rd, rs, rt	if (R[rs]＜R[rt]) 　R[rd]←1 else 　R[rd]←0	从 rs、rt 中取数后按带符号整数来判断两数大小 小于则 rd 中置 1，否则，rd 中清 0 (不进行溢出判断)
sltu rd, rs, rt	if (R[rs] ＜ R[rt]) 　R[rd]←1 else 　R[rd]←0	从 rs、rt 中取数后按无符号数来判断两数大小 小于则 rd 中置 1，否则，rd 中清 0 (不进行溢出判断)
ori rt, rs, imm16	R[rt]←R[rs]|ZeroExt(imm16)	从 rs 取数、将立即数 imm16 进行零扩展，然后两者按位或，结果送 rt
addiu rt, rs, imm16	R[rt]←R[rs]+SignExt(imm16)	从 rs 取数、将立即数 imm16 进行符号扩展，然后两者相加，结果送 rt(不进行溢出判断)
lw rt, rs, imm16	Addr←R[rs]+SignExt(imm16) R[rt]←M[Addr]	从 rs 取数、将立即数 imm16 进行符号扩展，然后两者相加，结果作为访存地址 从存储单元 Addr 中取数并送 rt
sw rt, rs, imm16	Addr←R[rs]+SignExt(imm16) M[Addr]←R[rt]	从 rs 取数、将立即数 imm16 进行符号扩展，然后两者相加，结果作为访存地址 从寄存器 rt 取数并送存储单元 Addr 中
beq rs, rt, imm16	Cond←R[rs]－R[rt] if (Cond eq 0) 　PC←PC+(SignExt(imm16)×4)	做减法以比较 rs 和 rt 中内容的大小 计算下条指令地址(根据比较结果，修改 PC) 转移目标地址采用相对寻址，基准地址为下条指令地址(即 PC+4)，位移量为立即数 imm16 经符号扩展后的值的 4 倍。所以，转移目标指令的范围为：相对于当前指令的前 32767 到后 32768 条指令
j target	PC＜31：2＞←PC＜31：28＞‖ target＜25：0＞	第一步无须进行 PC+4 而直接计算目标地址，符号‖表示“拼接”

6.2.2　数据通路的设计

为简化数据通路设计，假定所用的数据存储器和指令存储器皆为一种理想存储器。如图 6.14 所示，理想存储器有一个 32 位数据输入端 Data In；一个 32 位数据输出端 Data Out；还有一个读写公用的地址输入端 Address。控制信号有一个写使能信号 WE，写操作受时钟信号 Clk 的控制，假定采用下降沿触发，即在时钟下降沿开始写入信息。

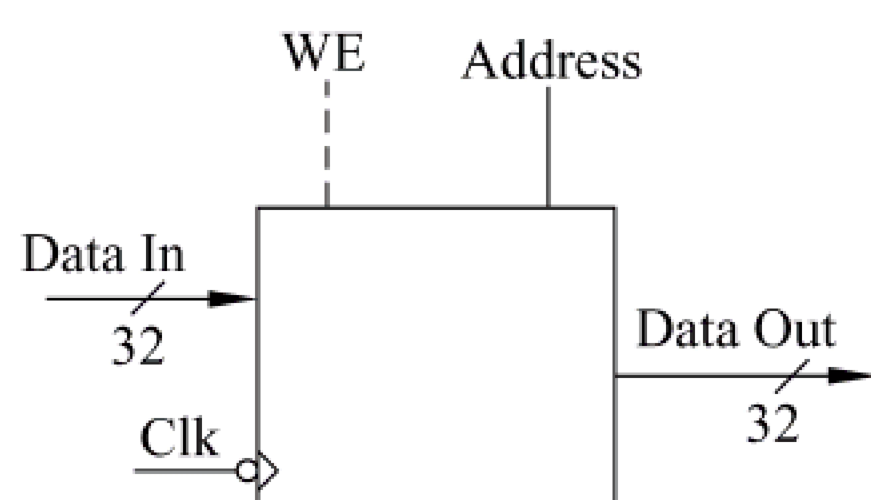


图 6.14　理想存储器外部结构

该理想存储器的读操作是组合逻辑操作，即在地址 Address 有效后，经过一个“取数时间”，数据输出端 Data

Out 上数据有效；写操作是时序逻辑操作，即在 WE 为 1 的情况下，当时钟 Clk 边沿到来时，Data In 开始写入存储单元中。

1. 算术逻辑部件的设计

上述 11 条指令涉及到带溢出判断的加法、减法、带符号整数的大小判断、无符号数的大小判断、相等判断以及各种逻辑运算等。为了支持这 11 指令包含的运算，ALU 必须具有相应的功能。

图 6.15 给出了一个实现上述 11 条指令中运算的 ALU。输入为两个 32 位操作数 A 和 B，其中，核心部件是能够进行加减运算的加法器，加法器的输出除了和/差 Add-Result 以外，还有进位标志 Add-carry、零标志 Zero、溢出标志 Add-Overflow 和符号标志 Add-Sign。有关加法器的实现可参见第 3 章内容。在操作控制端 ALUctr 的控制下，在 ALU 中执行"加法"、"减法"、"按位或"、"带符号整数比较小于置 1"和"无符号数比较小于置 1"等运算，Result 作为 ALU 运算的结果被输出，同时，零标志 Zero 和溢出标志 Overflow 也被作为 ALU 的结果标志信息输出。

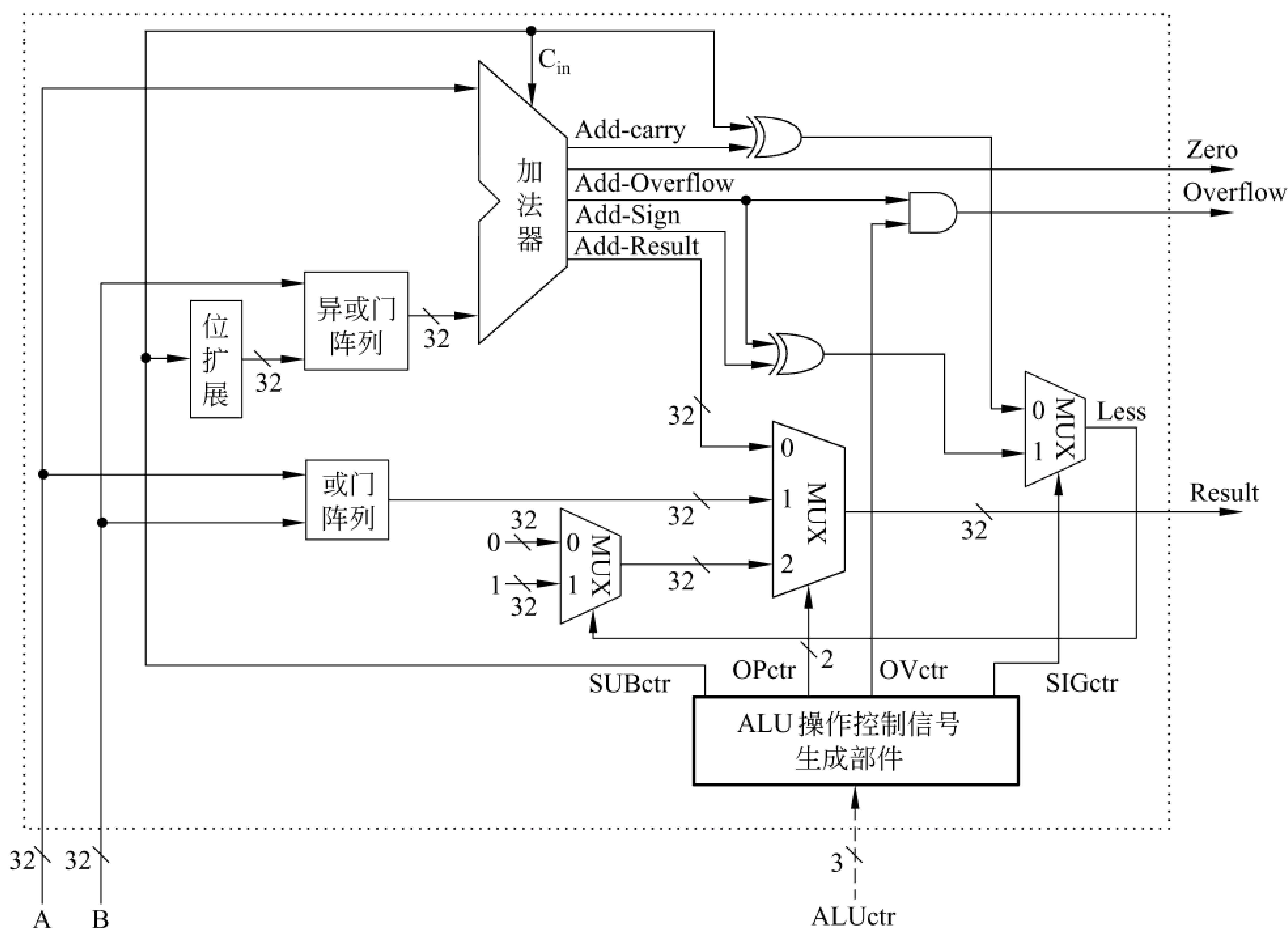


图 6.15　11 条目标指令的 ALU 实现

从图 6.15 可以看出，ALU 的操作由一个 ALU 操作控制信号生成部件产生的控制信号来控制，该控制逻辑的输入是 ALUctr 信号，输出有 4 个控制信号：①SUBctr 用来控制 ALU 执行加法还是减法运算，当 SUBctr＝1 时，做减法；当 SUBctr＝0 时，做加法；②OPctr 用来控制选择哪种运算的结果作为 Result 输出，因为 11 条目标指令中只有三种运算结果可能：加减、按位或和小于置 1，所以 OPctr 需要两位；③OVctr 用来控制是否要进行溢出判断，当 OVctr＝1 时，需要判断溢出，此时，若结果发生溢出，则溢出标志 Overflow

为 1；当 OVctr=0 时，无须判断溢出，此时，即使结果发生溢出，溢出标志 Overflow 也不为 1；④SIGctr 信号控制 ALU 是执行“带符号整数比较小于置 1”还是“无符号数比较小于置 1”功能，当 SIGctr=0，则执行“无符号数比较小于置 1”，当 SIGctr=1，则执行“带符号整数比较小于置 1”。

根据表 6.1 列出的每条指令的功能，可以了解到各条指令在 ALU 中所进行的运算，由此可列出各条指令对应的 4 种 ALU 操作控制信号取值，如表 6.2 所示。

表 6.2　11 条目标指令对应的 4 种 ALU 操作控制信号取值

指　令	功　　能	运 算 类 型	SUBctr	OVctr	SIGctr	OPctr
add rd，rs，rt	R[rd]←R[rs]+R[rt]	加(判溢出)	0	1	×	00
sub rd，rs，rt	R[rd]←R[rs]−R[rt]	减(判溢出)	1	1	×	00
subu rd，rs，rt	R[rd]←R[rs]−R[rt]	减(不判溢出)	1	0	×	00
slt rd，rs，rt	if (R[rs]<R[rt]) R[rd]←1 else R[rd]←0	减(不判溢出) 带符号整数比较大小	1	0	1	10
sltu rd，rs，rt	if (R[rs]<R[rt]) R[rd]←1 else R[rd]←0	减(不判溢出) 无符号数比较大小	1	0	0	10
ori rt，rs，imm16	R[rt]←R[rs] | ZeroExt (imm16)	按位或(不判溢出)	×	0	×	01
addiu rt，rs，imm16	R[rt]←R[rs]+SignExt (imm16)	加(不判溢出)	0	0	×	00
lw rt，rs，imm16	Addr←R[rs]+SignExt (imm16) R[rt]←M[Addr]	加(不判溢出)	0	0	×	00
sw rt，rs，imm16	Addr←R[rs]+SignExt (imm16) M[Addr]←R[rt]	加(不判溢出)	0	0	×	00
beq rs，rt，imm16	Cond←R[rs]−R[rt]	减(判 0)	1	0	×	××
	if (Cond eq 0) PC←PC+(SignExt(imm16)×4)	加(不判溢出)	0	0	×	00
j　target	PC<31：2>←PC<31：28> ‖ target<25：0>	无须 ALU 运算	×	×	×	××

注：×表示无论取什么值都不影响运算结果。

从表 6.2 可知，指令 addiu、lw、sw 和 beq 转移目标地址计算的 ALU 控制信号取值一样，都是进行加法运算并不判溢出，记为 addu 操作；指令 subu 和 beq 判 0 操作的 ALU 控制信号可看成一样，都做减法运算并不判溢出，记为 subu 操作。因此，这 11 条指令可以归纳为共有以下 7 种操作：add、sub、subu、slt、sltu、or、addu，需要三位对其进行编码，因而 ALU 的操作控制输入端 ALUctr 至少有三位。

在对 ALUctr 进行编码时，可以根据这些 ALU 操作和 4 种 ALU 操作控制信号的对应关系进行优化，例如，把加减控制(SUBctr)、溢出判断(OVctr)和符号控制(SIGctr)等信号分别对应到不同的位来进行控制。表 6.3 给出了 ALUctr 的一种三位编码方案。

表 6.3　ALUctr 的三位编码及其对应的操作类型和 ALU 控制信号

ALUctr＜2：0＞	操作类型	SUBctr	OVctr	SIGctr	OPctr＜1：0＞	OPctr 的含义
0　0　0	addu	0	0	×	0 0	选择加法器的结果输出
0　0　1	add	0	1	×	0 0	选择加法器的结果输出
0　1　0	or	×	0	×	0 1	选择“按位或”结果输出
0　1　1	(未用)					
1　0　0	subu	1	0	×	0 0	选择加法器的结果输出
1　0　1	sub	1	1	×	0 0	选择加法器的结果输出
1　1　0	sltu	1	0	0	1 0	选择小于置位结果输出
1　1　1	slt	1	0	1	1 0	选择小于置位结果输出

根据表 6.3 得到各输出控制信号的逻辑表达式如下：

```
SUBctr=ALUctr<2>
OVctr=!ALUctr<1>&ALUctr<0>
SIGctr=ALUctr<0>
OPctr<1>=ALUctr<2>&ALUctr<1>
OPctr<0>=!ALUctr<2>ALUctr<1>&!ALUctr<0>
```

根据上述逻辑表达式，不难实现图 6.15 中的 ALU 操作控制信号生成部件。

如果要实现更多指令，则 ALU 必须支持更多的运算，如取负(neg)、取反(not)、与(and)、异或(xor)、或非(nor)等，如果在 ALU 中考虑所有这些情况的话，需在图 6.15 所示的 ALU 中增加相应的取负、按位取反、按位与、按位异或、按位或非等逻辑电路，同时 ALU 输出结果选择控制信号 OPctr 的位数需扩充到至少三位，ALUctr 的位数需扩充到 4 位。表 6.4 给出了 ALUctr 的一种 4 位编码方案，其中，ALUctr＜3＞对应加减控制(SUBctr)，ALUctr＜0＞对应溢出判断/符号控制(OVctr/SIGctr)。

表 6.4　ALUctr 的 4 位编码及其对应的操作类型和 ALU 控制信号

ALUctr＜3：0＞	操作类型	SUBctr	OVctr	SIGctr	OPctr＜2：0＞	OPctr 的含义
0　0　0　0	addu	0	0	×	0　0　0	选择加法器的结果输出
0　0　0　1	add	0	1	×	0　0　0	选择加法器的结果输出
0　0　1　0	and	×	0	×	0　1　0	选择“按位与”结果输出
0　0　1　1	or	×	0	×	0　1　1	选择“按位或”结果输出
0　1　0　0	not	×	0	×	1　0　0	选择“按位反”结果输出
0　1　0　1	nor	×	0	×	1　0　1	选择“按位或非”输出

续表

ALUctr<3:0>	操作类型	SUBctr	OVctr	SIGctr	OPctr<2:0>	OPctr 的含义
0 1 1 0	xor	×	0	×	1 1 0	选择"按位异或"输出
0 1 1 1	neg	×	0	×	1 1 1	选择"取负"结果输出
1 0 0 0	subu	1	0	×	0 0 0	选择加法器的结果输出
1 0 0 1	sub	1	1	×	0 0 0	选择加法器的结果输出
1 0 1 0	sltu	1	0	0	0 0 1	选择小于置位结果输出
1 0 1 1	slt	1	0	1	0 0 1	选择小于置位结果输出

注：若再加上其他操作功能，如左移、右移、前导0个数、前导1个数等，ALU结果选择操作控制信号OPctr的位数和该表的表项都还需要进一步扩充。

2. 取指令部件的设计

从上述指令功能的RTL描述中可以看出，每条指令的第一步是公共操作，完成取指令并计算下条指令地址的功能。因此，在数据通路中，需要专门设计一个取指令部件来完成上述功能。

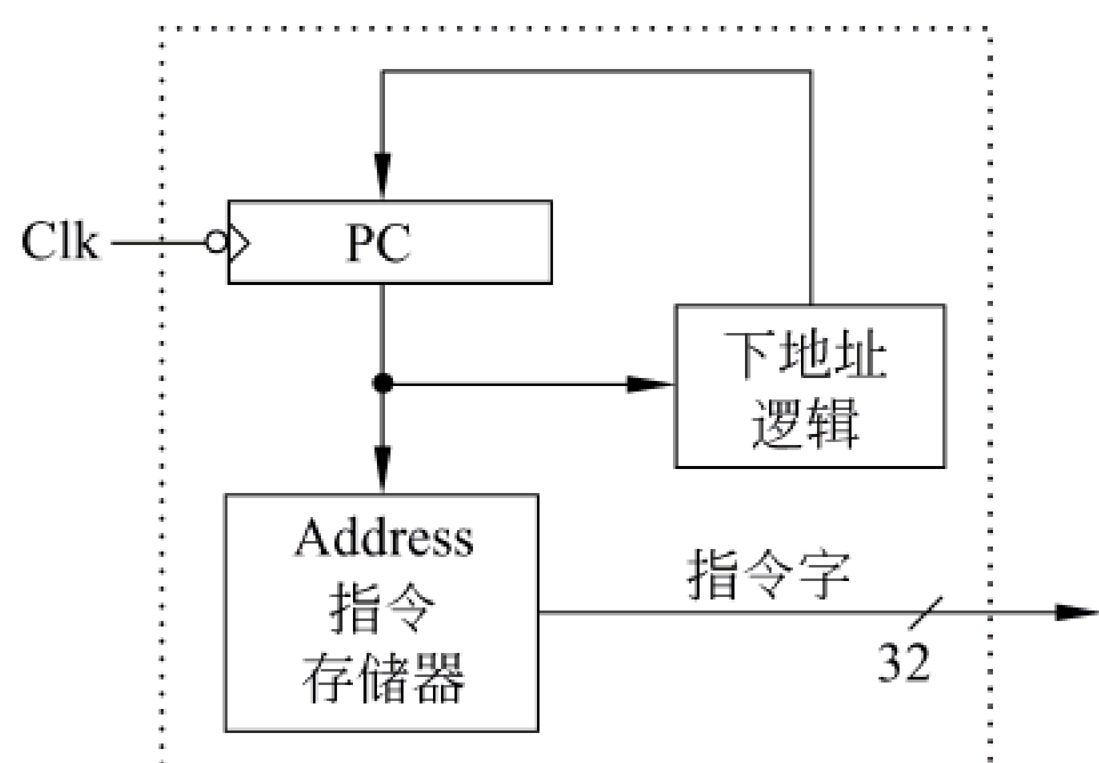


图6.16　取指令部件示意图

图6.16是取指令部件的示意图。假定指令专门存放在指令存储器中，它只有读操作，读指令操作可看成是组合逻辑操作，所以无须控制信号的控制，只要给出指令地址，经过一定的"取数时间"后，指令被送出。指令的地址来自PC，有专门的下地址逻辑来计算下条指令的地址，然后送PC。因为是单周期处理器，每个时钟周期执行一条指令，所以，每来一个时钟，PC的值都被更新一次，因而，PC无须"写使能"信号控制。下地址逻辑中，要区分是顺序执行还是转移执行。若是顺序执行，则执行PC+4；若是转移执行，则要根据当前指令是分支指令还是跳转指令来计算转移目标地址。

3. R-型指令的数据通路

R-型指令都涉及到对Rs和Rt内容的运算，最终要把ALU的运算结果送目的寄存器Rd。像add和sub等指令还要判断结果是否溢出，只有不溢出时才写结果到Rd，否则转异常处理程序执行。

图6.17是R-型指令相关的数据通路示意图，用它可以完成对两个寄存器Rs和Rt内容的运算并将结果写入Rd寄存器。

指令中Rs和Rt字段是两个源操作数寄存器的编号，Rd字段是目的寄存器编号，所以，寄存器堆的两个输出地址端Ra和Rb分别与Rs和Rt字段相连，输入地址端Rw与Rd字段相连。ALU运算结果连到寄存器堆的输入端busW，控制信号RegWr为"写使能"信号，只有在RegWr信号为1并且不溢出的情况下，运算结果才能写入寄存器堆，显然R-型指令执行时，RegWr信号应该为1。

实现目标中有5条R-型指令：add、sub、subu、slt和sltu，根据表6.2可知，它们对应

ALU 的 5 种操作：add、sub、subu、slt 和 sltu，因此，可根据不同的指令，控制将不同的操作编码送到 ALU 操作控制端 ALUctr，以使在 ALU 中进行不同指令所对应的运算。

图 6.17　支持 R-型指令功能的数据通路

当前时钟周期内执行的运算结果总是在下一个时钟到来时，开始写到寄存器堆中。为了能写入正确的稳定结果，ALU 操作控制信号必须保持稳定。

4. 立即数运算指令的数据通路

I-型带立即数的运算类指令都涉及到对 16 位立即数进行符号扩展或零扩展，然后和 Rs 的内容进行运算，最终把 ALU 的运算结果送目的寄存器 Rt。

11 条目标指令中 ori 和 addiu 指令是 I-型带立即数的运算类指令，涉及到的操作有 or 和 addu。图 6.18 是在图 6.17 的基础上增加了 I-型立即数运算类指令功能而得到的数据通路示意图，因此，它同时也能完成 R-型指令的执行。与图 6.17 相比，有三处变动。(1)因为 R-型指令和 I-型指令的目的寄存器不同，所以在寄存器堆的输入地址端 Rw 处，增加了一个多路选择器，由控制信号 RegDst 控制选择 Rd 为目的寄存器还是 Rt 为目的寄存器；(2)因为 I-型指令的立即数只有 16 位，需要对其扩展为 32 位才能送到 32 位 ALU 运算。对于按位逻辑运算，应采用零扩展，对于算术运算，应采用符号扩展。因此，在数据通路中应增加一个扩展器 Extender，由控制信号 ExtOp 控制进行符号扩展还是零扩展，其输入端连到指令的 imm16 字段；(3)因为 R-型指令和 I-型指令在 ALU 的 B 口的操作数来源不同，所以，在

图 6.18　支持 I-型带立即数运算指令功能的数据通路

B 输入端增加了一个多路选择器，由控制信号 ALUSrc 控制选择 busB 还是扩展器输出作为 ALU 的 B 口操作数。

5. Load/Store 指令的数据通路

lw/sw 指令是 I-型指令，lw 指令的功能为 R[Rt]←M[R[Rs]+SignExt(imm16)]；sw 指令的功能为 M[R[Rs]+SignExt(imm16)]←R[Rt]。Load 指令和 Store 指令的地址计算过程一样，都要先对立即数 imm16 进行符号扩展，然后和寄存器 Rs 的内容相加，得到访存地址。Load 指令从该地址中读取一个 32 位数，送到寄存器 Rt 中；Store 指令则相反。

图 6.19 是在图 6.18 的基础上增加了 Load/Store 指令功能而得到的数据通路示意图，因此，它同时也能完成 R-型和 I 型运算类指令的执行。与图 6.18 相比，有两处变动。(1)因为运算类指令和 Load 指令写入目的寄存器的结果的来源不同，所以在寄存器堆的输入数据端 busW 处，增加了一个多路选择器，由控制信号 MemtoReg 控制选择将 ALU 结果还是存储器读出数据写入目的寄存器；(2)因为 Load/Store 指令需要读写数据存储器，故增加了数据存储器。访存地址在 ALU 中计算，因此数据存储器的地址端 Adr 连到 ALU 的输出。Store 指令将 Rt 内容送存储器，所以直接将 busB 连到数据存储器的 Data In 输入端，而将输出端 Data out 连到 busW 端的多路选择器上。控制信号 MemWr 用作“写使能”信号。Load/Store 指令的地址运算对立即数 imm16 进行符号扩展，ALUctr 输入端的操作类型是不判溢出的加法 addu。

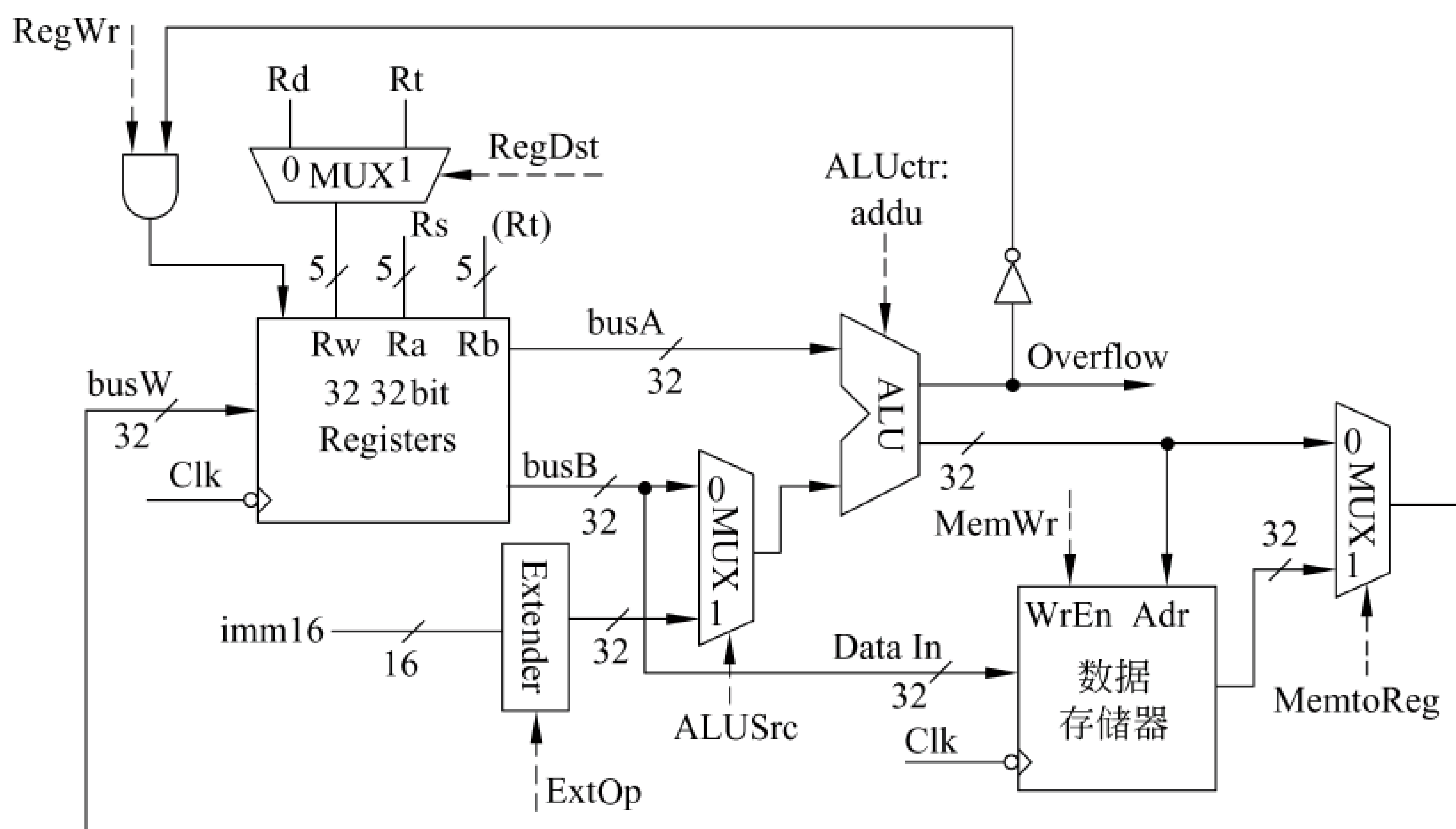


图 6.19 支持 Load/Store 指令功能的数据通路

6. 分支指令的数据通路

分支指令也是 I-型指令，能根据不同的条件进行分支转移。例如，相等转移指令 beq 的功能为 if (R[Rs]=R[Rt]) PC←PC+4+(SignExt(imm16)×4) else PC←PC+4。图 6.20 是在图 6.19 基础上增加 beq 指令功能而得到的数据通路。与图 6.19 相比，主要增加了取指令部件，转移目标地址计算在下地址逻辑中实现，在 ALU 中执行的是不判溢出的减法操作 subu。

下地址逻辑的输出是下条指令地址，4 个输入是 PC、Zero 标志、立即数 imm16 和控制信号 Branch。在 ALU 中对 R[Rs]和 R[Rt]做减法得到一个 Zero 标志，根据 Zero 标志可

判断是否转移。转移目标地址的计算需要先对立即数 imm16 进行符号扩展再乘 4,然后和基准地址 PC+4 相加。所以,ALU 的输出 Zero 标志需送到下地址逻辑,立即数 imm16 和 PC 也要送到下地址逻辑,控制信号 Branch 表示当前指令是否是分支指令,也应送到下地址逻辑,以决定是否按分支指令方式计算下条指令地址。

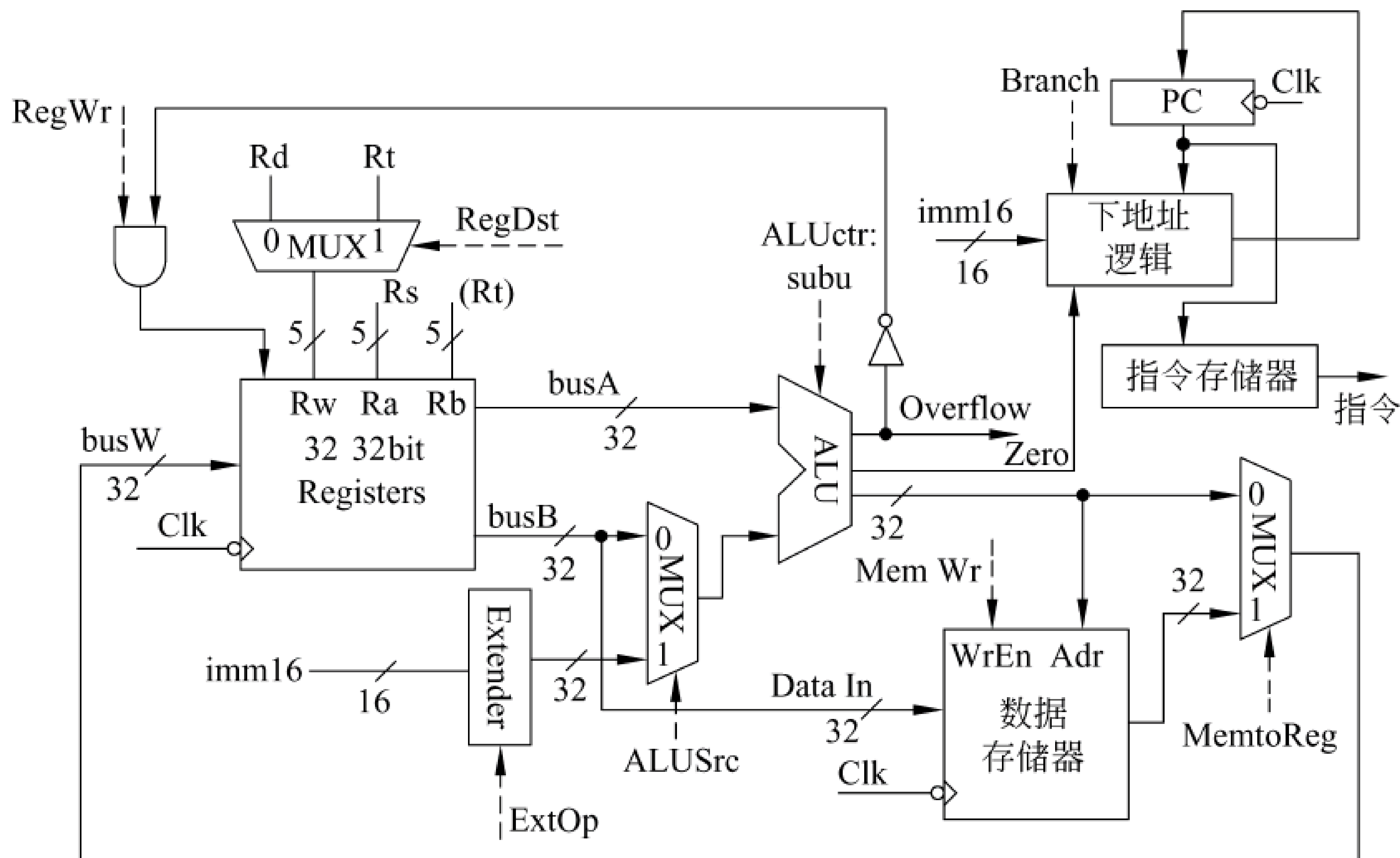


图 6.20　支持分支指令功能的数据通路

因为指令长度为 32 位,主存按字节编址,所以指令地址总是 4 的倍数,即最后两位总是 00,因此,PC 中只需存放前 30 位地址,即 PC<31:2>。这样,下条指令地址的计算方法如下。

顺序执行时:PC<31:2>←PC<31:2>+1。

转移执行时:PC<31:2>←PC<31:2>+1+SignExt[Imm16]。

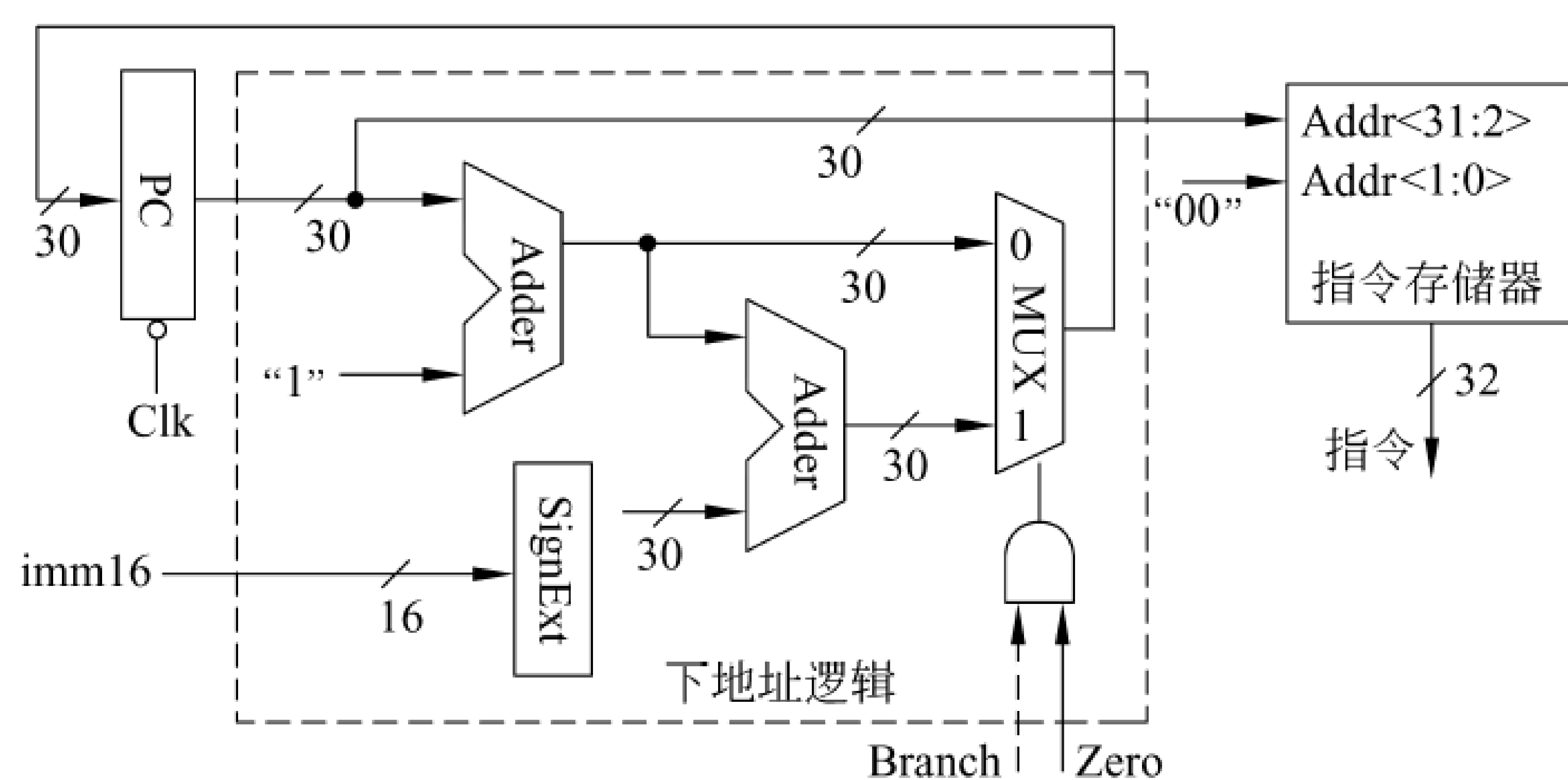
取指令时:指令地址=PC<31:2> ‖ “00”。

实现上述下地址逻辑有两种方式。图 6.21(a)是一种快速但昂贵的方式,而图 6.21(b)则相反。因为,在图 6.21(b)的电路中,只能等到 Zero 有值后才能进行地址计算,而在图 6.21(a)的电路中,只要指令译码结果不是分支指令(Branch=0),则立即可以得到下条指令地址为 PC+1。但是采用图 6.21(b)的慢速方式对性能没有影响,因为单周期处理器的时钟周期以最长指令周期为准,Load 指令需要访问内存,分支指令再慢也比 Load 指令快。图 6.21(b)中用低位进位为 1 来实现“+1”操作,节省了一个加法器(Adder)。

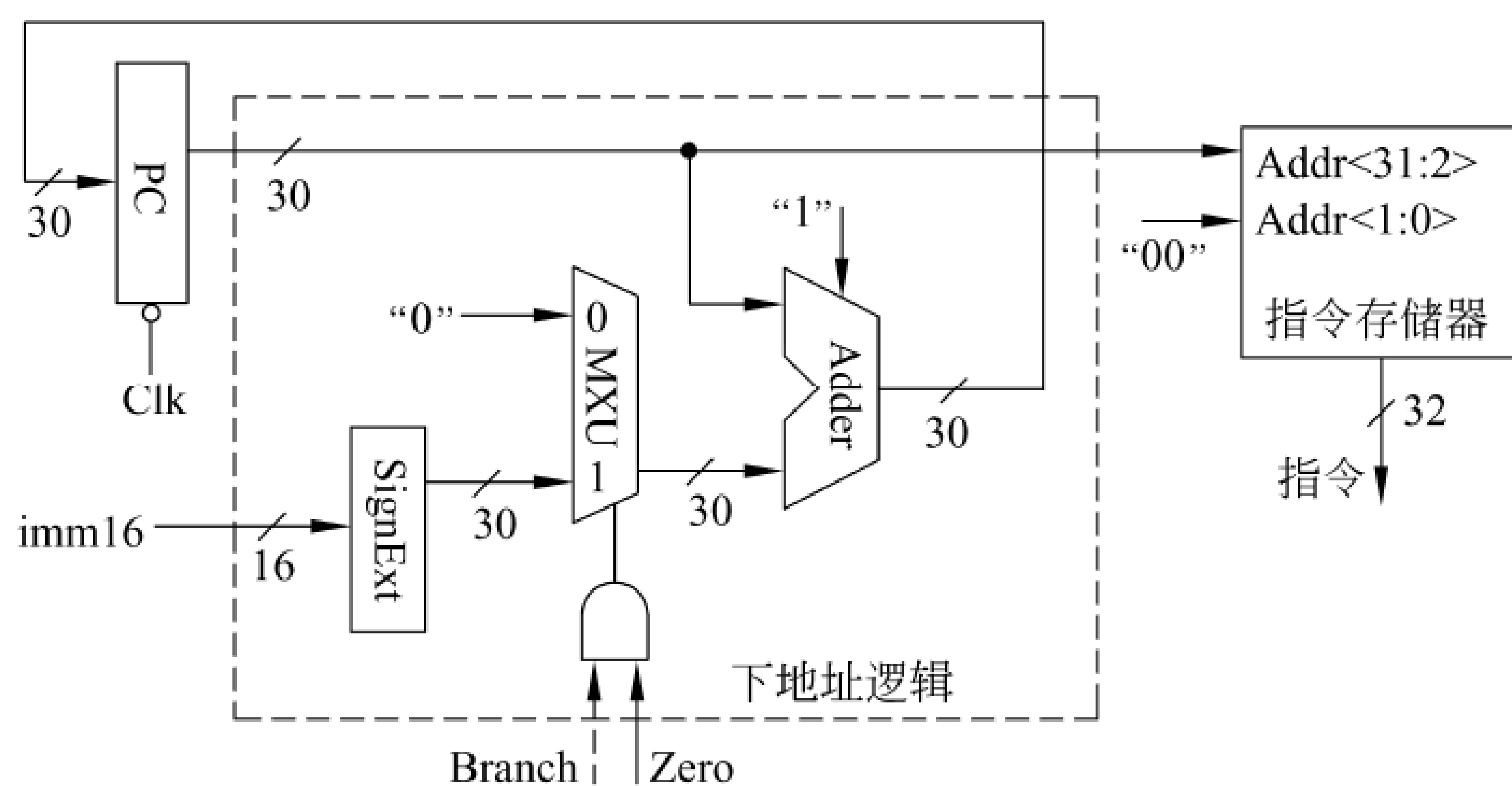
从图 6.21 可看出,每来一个时钟 Clk,当前 PC 作为指令地址被送到指令存储器去取指令,同时下地址逻辑计算下条指令地址,送到 PC 的输入端,在下个时钟到来后才写入 PC。

7. 无条件转移指令的数据通路

无条件转移指令是 J-型指令,指令中给出了 26 位目标地址,其功能是无条件将目标地址设置到 PC 中。跳转目标地址的具体计算方法为 PC<31:2>←PC<31:28> ‖ target<25:0>。图 6.22 给出了在图 6.21 的基础上加上无条件转移指令功能的完整的取指令部件示意图。下地址逻辑中增加了跳转目标地址的计算功能,并通过控制信号 Jump 来选择作为下条指令的 PC 值。



(a) 快速但昂贵的设计方案



(b) 慢速但节省的设计方案

图 6.21 分支指令的下地址逻辑设计

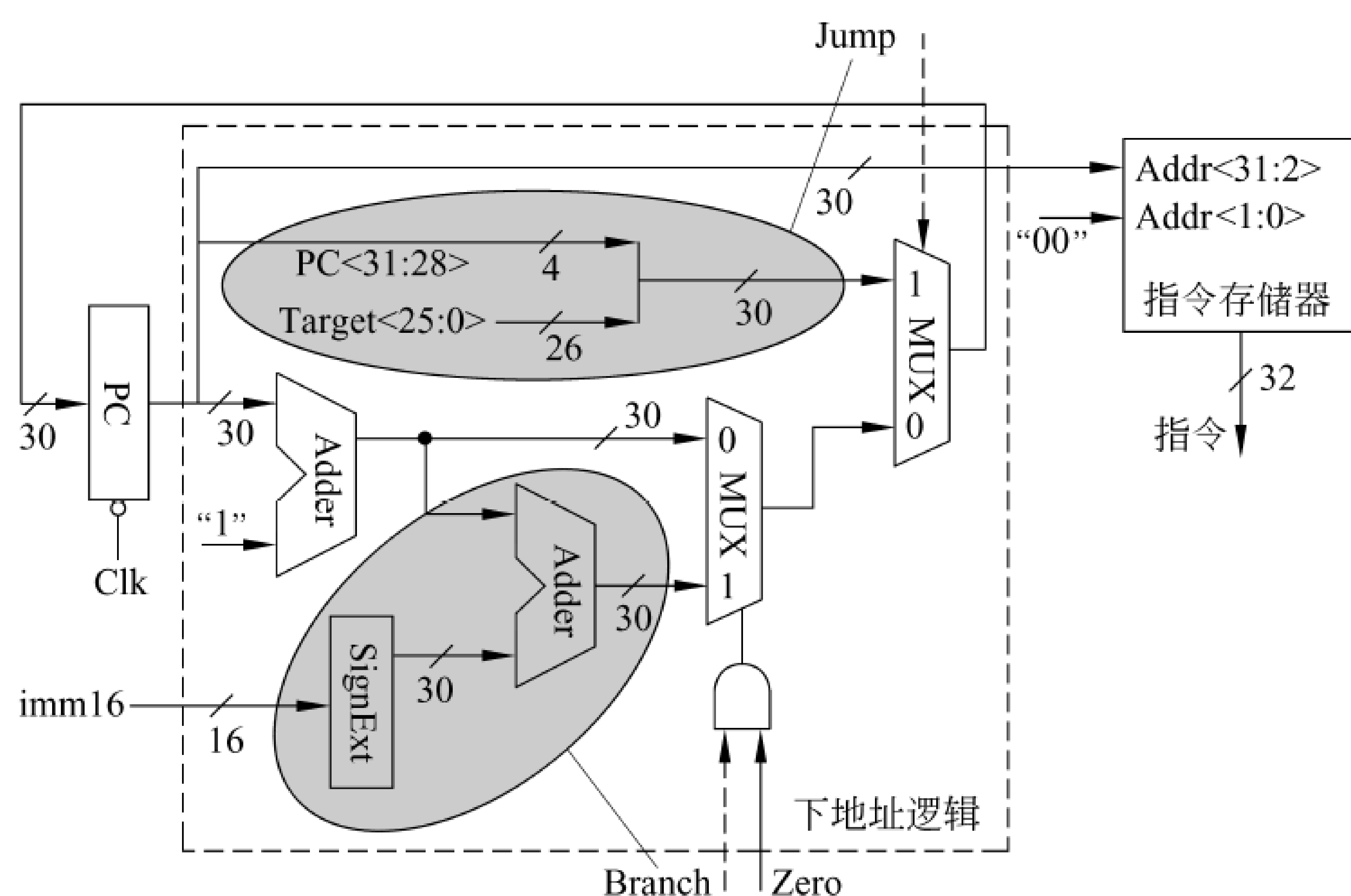


图 6.22 完整的取指令部件

在取指令阶段开始，新指令还没被取出、译码，所以取指令部件中的控制信号(Branch、Jump)的值还是上条指令产生的旧值，此外，新指令还没被执行，所以标志(Zero)也为旧值。但是，由这些旧控制信号值确定的地址只被送到 PC 输入端，不会写入 PC，所以不会影响取

指令功能。只要保证在下个时钟 Clk 到来之前能产生正确的下条指令地址即可。单周期方式下，下个时钟会在足够长的时间(最长的指令周期)后到来，此时，控制信号早就是新取出的当前指令对应的控制信号了，因而，取指令部件能得到正确的下条指令地址，在下个时钟到来前被送到 PC 的输入端，当下个时钟到来时，该地址开始被写入 PC，并作为指令地址从指令存储器中取出指令。

由图 6.22 可知，取指令部件的输出是指令，输入有三个：标志 Zero 和控制信号 Branch、Jump。下地址逻辑中的立即数 imm16 和目标地址 target＜25：0＞都直接来自取出的指令。因此，取指令部件的外部结构如图 6.23 所示。图中所示的取出指令 Instruction＜31：0＞表示 I-型或 R-型指令格式的部分字段。

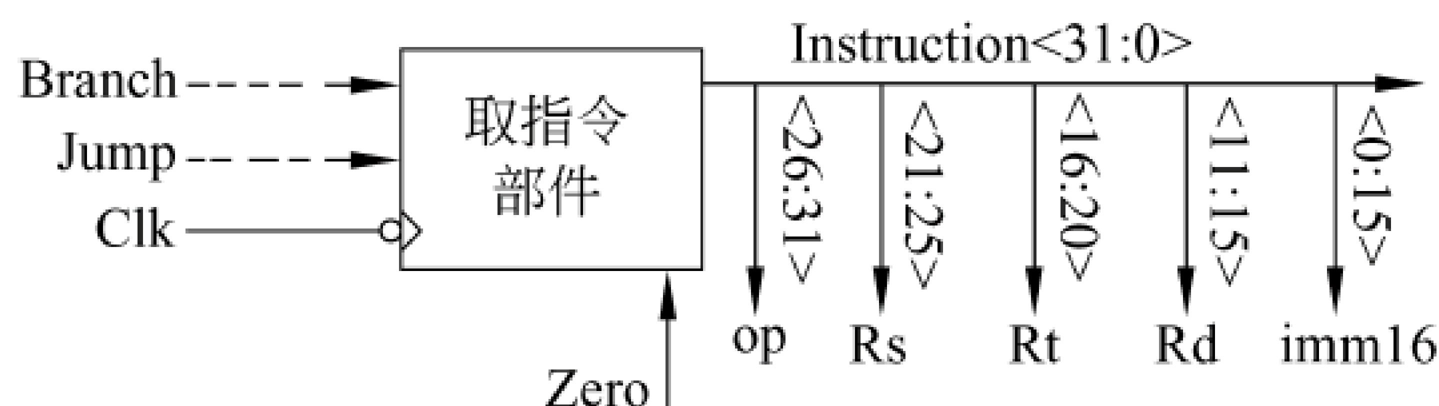


图 6.23　取指令部件的外部结构

8. 综合 11 条指令的完整数据通路

综合考虑上述所有数据通路的结构，可得到如图 6.24 所示的完整单周期数据通路。图中所有加下划线的都是控制信号名称，用虚线表示。指令执行结果总是在下个时钟到来时开始保存在寄存器堆、数据存储器或 PC 中。

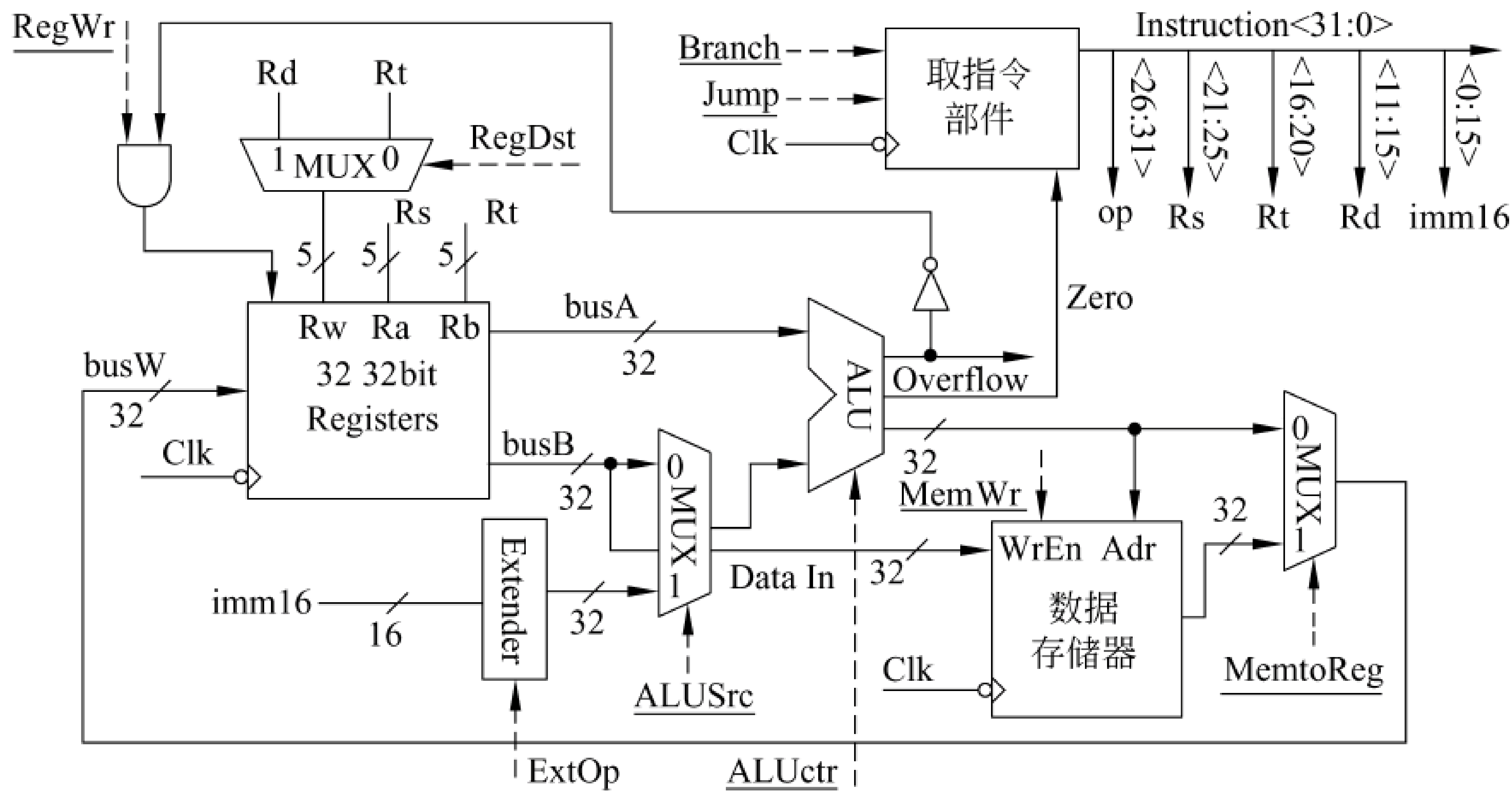


图 6.24　完整的单周期数据通路

至此，已完成了所有 11 条指令所用到的数据通路，包括所用元件及其互连，并给出了控制信号。下一步应考虑如何产生控制信号，这就是控制逻辑单元的设计。

6.2.3　控制逻辑单元的设计

控制单元主要包含一个指令译码器，输入的是指令操作码 op(R-型指令还包括功能码 func)，输出的是控制信号。所以，控制单元的设计过程如下。

（1）根据每条指令的功能，分析控制信号的取值，并在表中列出。

（2）根据列出的指令和控制信号的关系，写出每个控制信号的逻辑表达式。

1. 控制信号取值分析

根据对取指令阶段的执行情况的分析，可知，Clk 信号到来后，经锁存延迟（Clk-to-Q）时间，PC 的值作为访存地址被送到指令存储器，经过“取数时间”后，指令被取出，后送控制单元，经指令译码器译码，送出控制信号。随后，每条指令便在控制信号的控制下，完成相应的功能。以下分析每条指令执行阶段控制信号的取值情况。

（1）R-型指令执行阶段

图 6.25 是 R-型指令的执行过程示意图，其中的粗线描述了 R-型指令的数据在数据通路中的执行路径：Register File（Rs，Rt）→busA，busB→ALU→Register File（Rd）。

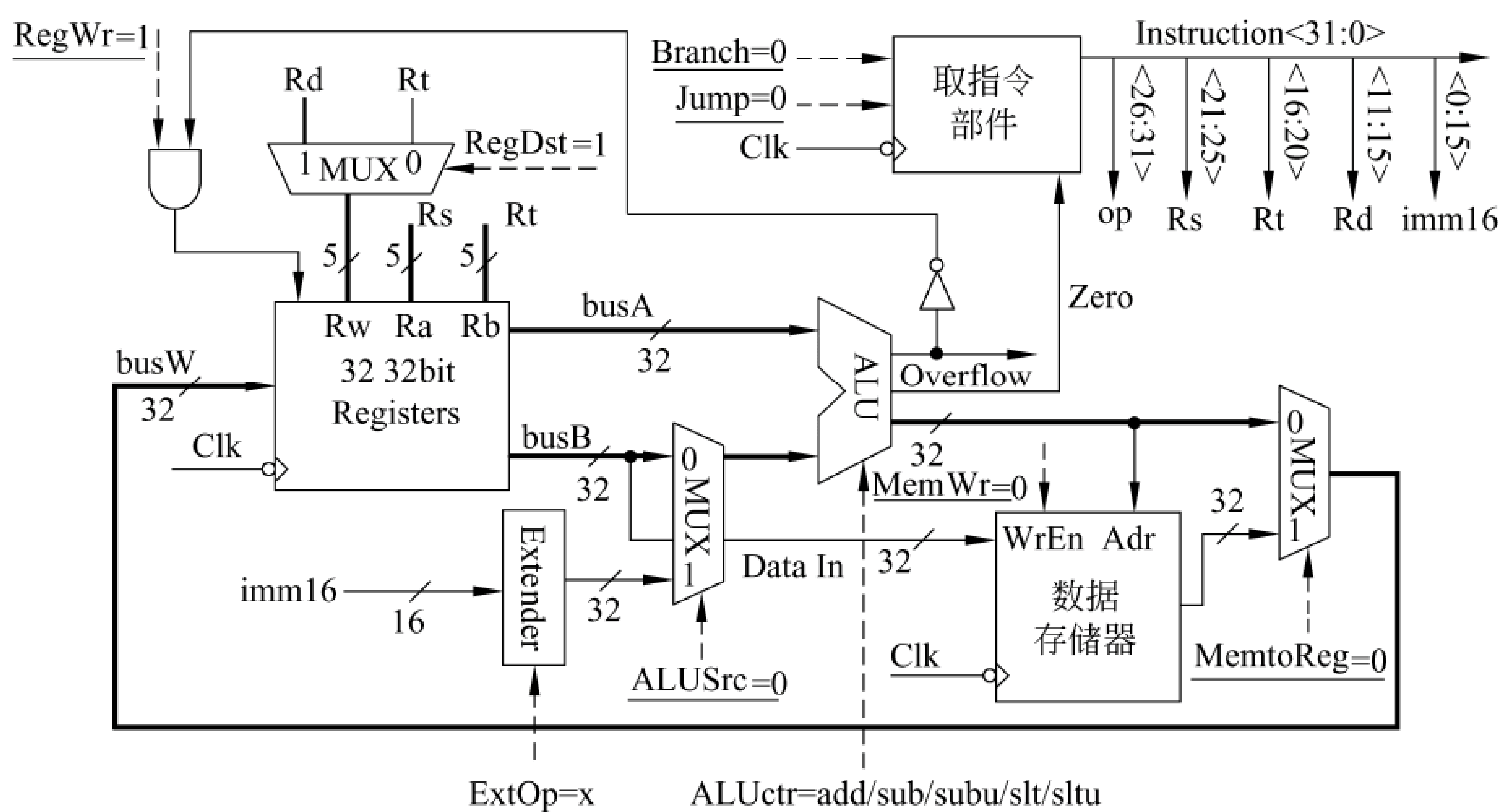


图 6.25　R-型指令执行过程

控制信号的取值分析如下。

Branch＝Jump＝0：因为是非分支指令、非无条件跳转指令。

RegDst＝1：因为 R-型指令的目的寄存器为 Rd。

ALUSrc＝0：保证选择 busB 作为 ALU 的 B 口操作数。

ALUctr＝add/sub/subu/slt/sltu：5 条 R-型指令的操作都不同，因此对应 5 种类型。

MemtoReg＝0：保证选择 ALU 结果送到目的寄存器。

RegWr＝1：保证在下个时钟到来时，在不发生溢出的情况下结果被写到目的寄存器。

MemWr＝0：保证在下个时钟到来时，不会有信息写到数据存储器。

ExtOp＝x：因为 ALUSrc＝0，所以扩展器 Extender 的值不会影响执行结果，故 ExtOp 取 0 或 1 都无所谓。

图 6.26 给出了 R-型指令的操作定时过程。从图中可以看出，下条指令地址 PC＋4 将在下个 Clk 到来时开始写入 PC，指令执行结果（ALU 输出）也将在下个 Clk 到来时开始写入目的寄存器 Rd。

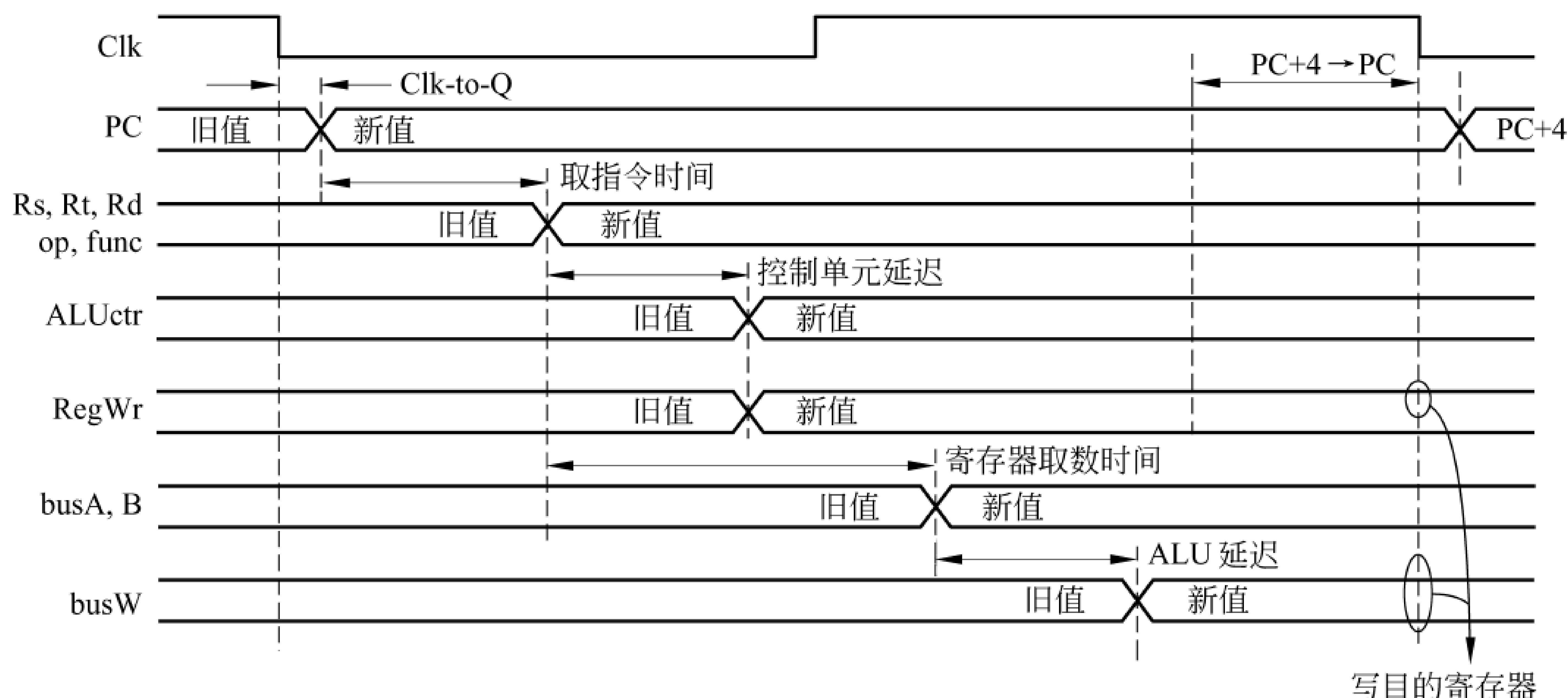


图 6.26　R-型指令的操作定时

(2) I-型运算指令执行阶段

图 6.27 中的粗线给出了 I-型运算指令的执行过程，从图中可以看出其数据在数据通路中的执行路径为 Register File(Rs)→busA，Extender(imm16)→ALU→Register File(Rt)。

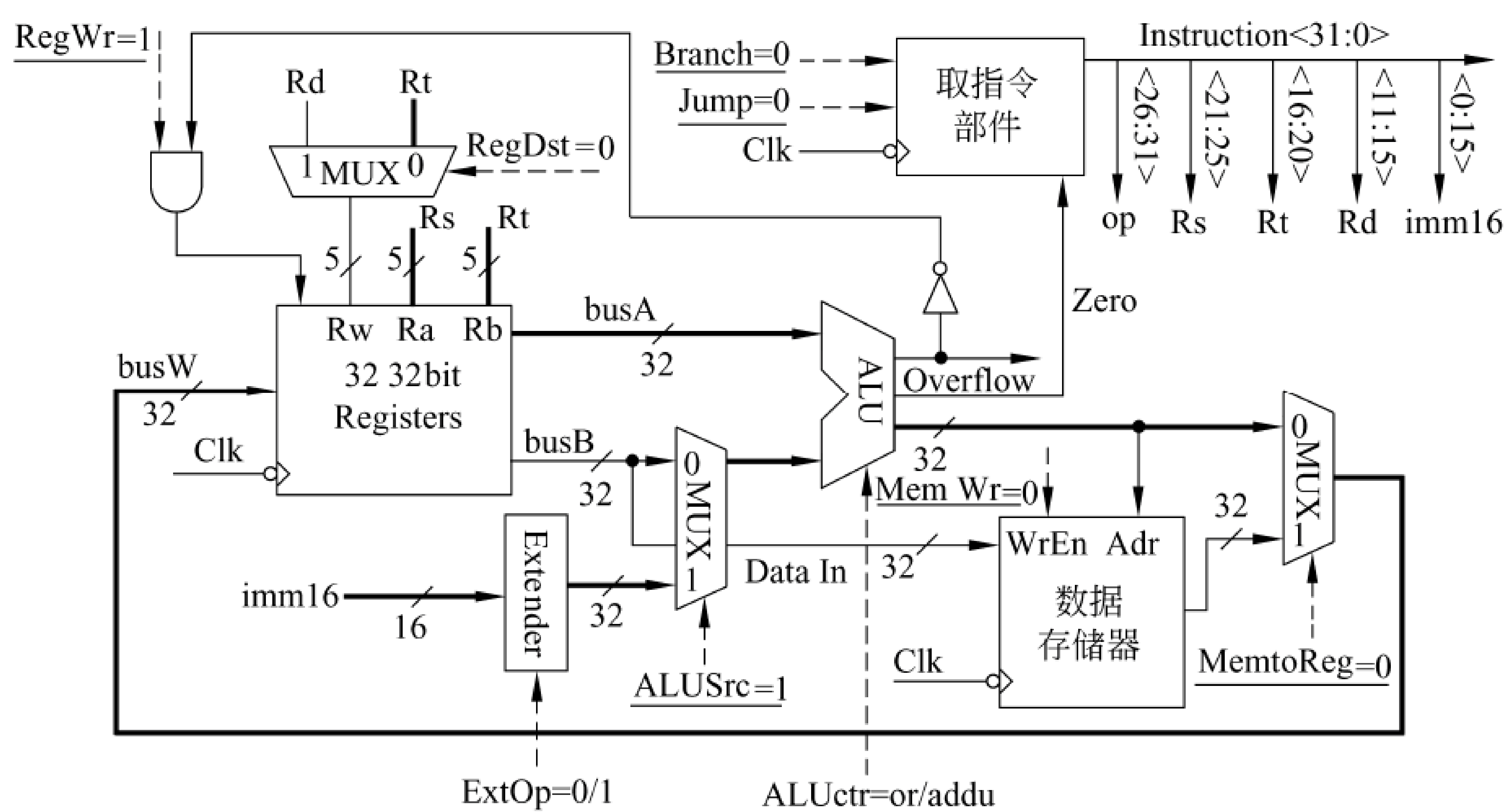


图 6.27　I-型运算指令执行过程

对于目标指令中的 I-型运算指令 ori 和 addiu，不难看出各控制信号的取值如下。

Branch=Jump=0，RegDst=0，ALUSrc=1，ALUctr=or/addu(ori 指令为 or，addiu 指令为 addu)，MemtoReg=0，RegWr=1，MemWr=0，ExtOp=0/1(ori 指令为 0，addiu 指令为 1)。

(3) Load/Store 指令执行阶段

图 6.28 中的粗线给出了 Load 指令的执行过程，从图中可以看出其数据在数据通路中的执行路径为 Register File(Rs)→busA，Extender(imm16)→ALU(addu)→数据存储器→RegisterFile(Rt)。

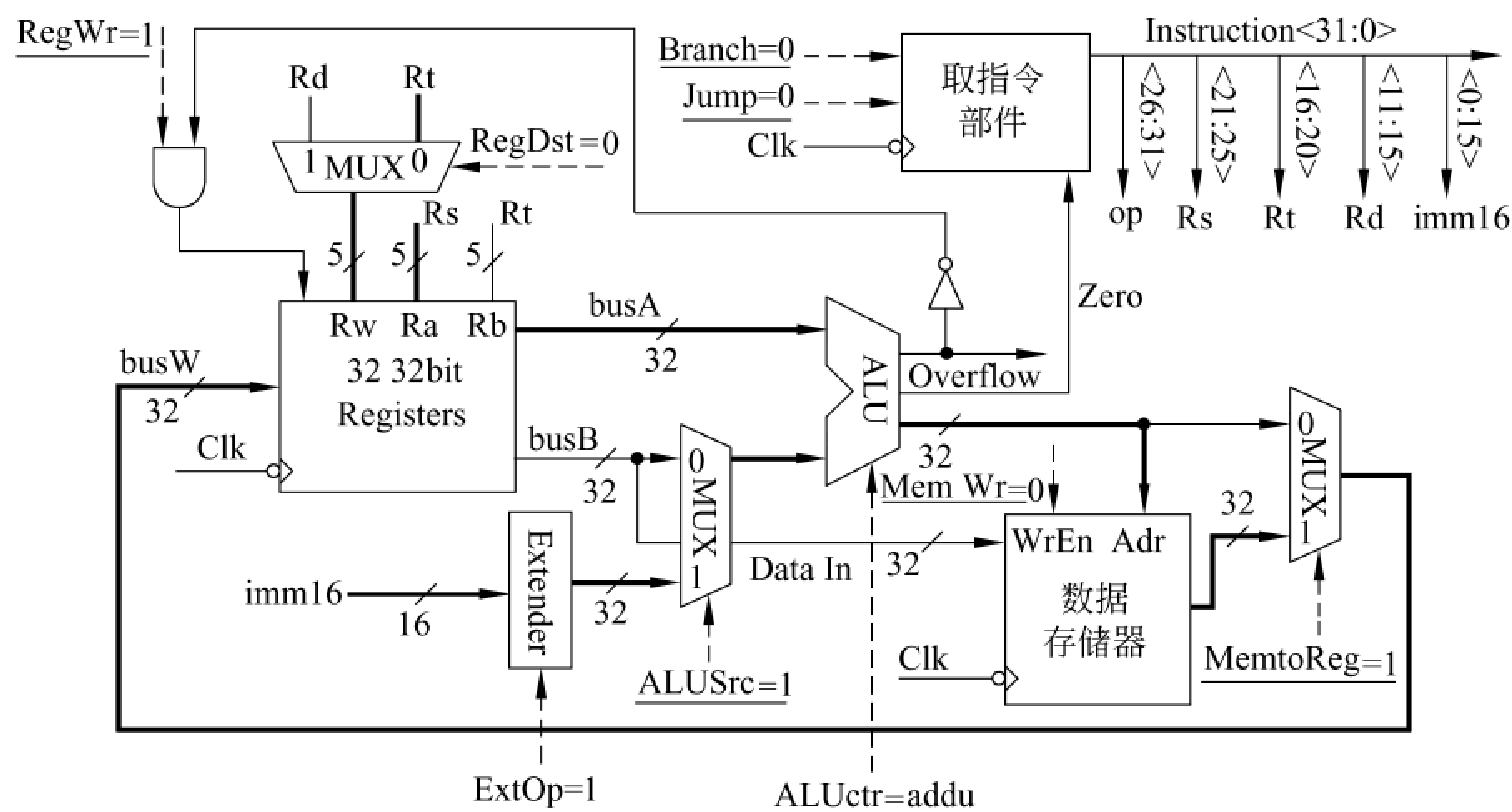


图 6.28　Load 指令的执行过程

对于 Load 指令，不难看出各控制信号的取值如下：Branch＝Jump＝0，RegDst＝0，ALUSrc＝1，ALUctr＝addu，MemtoReg＝1，RegWr＝1，MemWr＝0，ExtOp＝1(符号扩展)。

图 6.29 是 Store 指令的执行过程示意图。从图中可看出，其数据在数据通路中的执行路径为 Register File(Rs，Rt)→busA，Extender(imm16)，busB→ALU(addu)，busB→数据存储器。

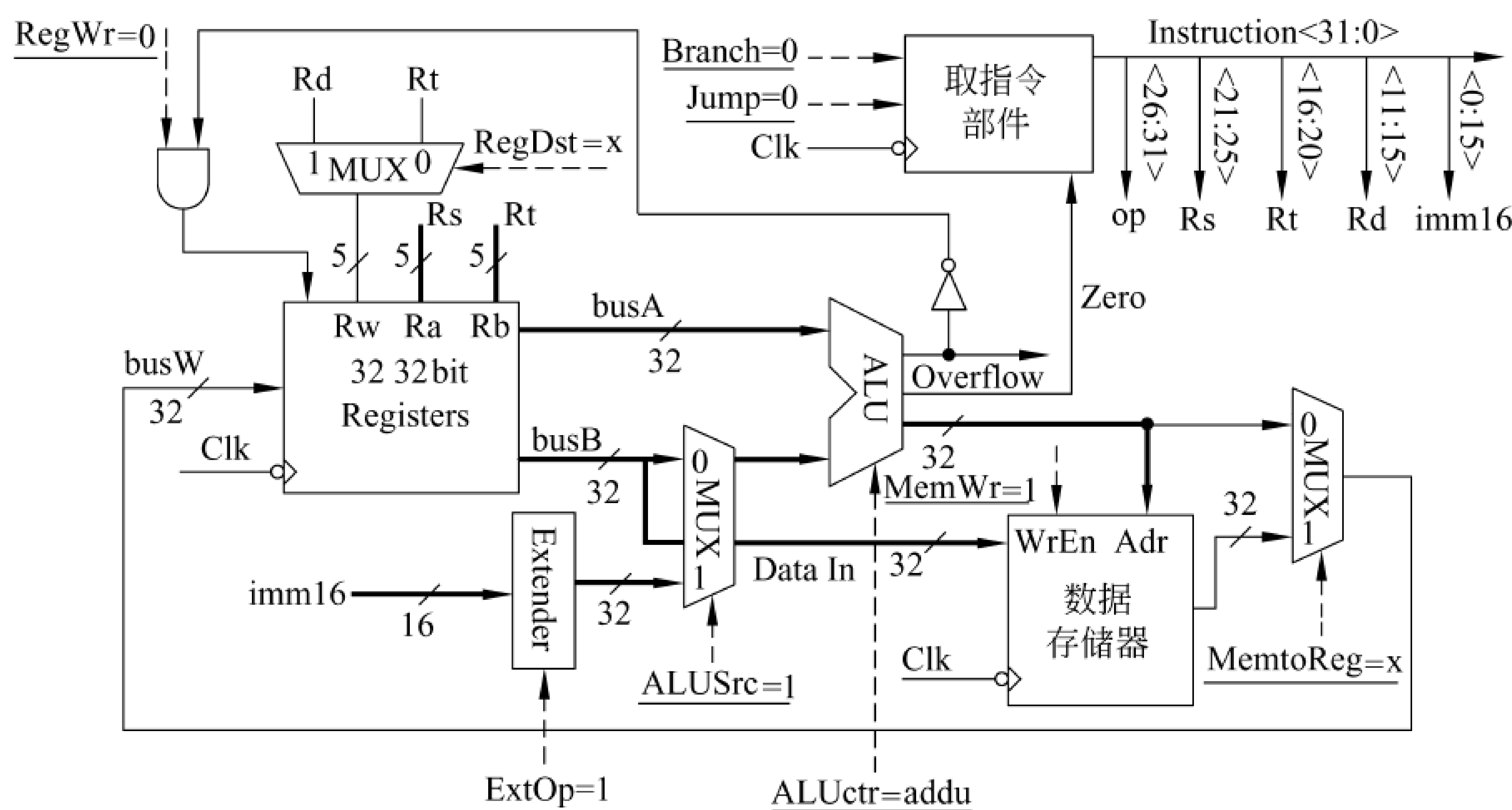


图 6.29　Store 指令的执行过程

根据 Store 指令的功能，不难看出各控制信号的取值如下：Branch＝Jump＝0，RegDst＝x，ALUSrc＝1，ALUctr＝addu，MemtoReg＝x，RegWr＝0，MemWr＝1，ExtOp＝1（符号扩展）。

（4）分支指令执行阶段

分支指令 beq 通过做减法来得到 Zero 标志，然后将 Zero 标志送到取指令部件，与控制信号 Branch 一起进行“与”操作，来控制下条指令地址的计算。因此，其数据在数据通路中的执行路径为 Register File(Rs，Rt)→busA，busB→ALU(subu)→Zero 标志→取指令部件。在取指令部件(参见图 6.22)中，若 Zero＝0，则执行 PC<31∶2>←PC<31∶2>＋1；若 Zero＝1，则执行 PC<31∶2>←PC<31∶2>＋1＋SignExt[imm16]。

根据 beq 指令的功能，不难看出各控制信号的取值如下：Branch＝1，Jump＝0，RegDst＝x，ALUSrc＝0，ALUctr＝subu，MemtoReg＝x，RegWr＝0，MemWr＝0，ExtOp＝x。

（5）无条件转移指令执行阶段

在 11 条指令中，无条件跳转指令最简单，除了改变 PC 的值外，不做任何其他工作。因此，只在取指令部件中由控制信号 Jump 控制下条指令地址的计算。

Jump 指令的控制信号取值只要保证：①Branch＝0，Jump＝1，以使取指令部件能正确得到下条指令地址；②RegWr＝0，MemWr＝0，以使寄存器堆和数据存储器在本指令执行时不被写入任何结果。其他控制信号的取值无所谓。

综上所述，得到各指令的控制信号取值，如表 6.5 所示。

表 6.5　各指令控制信号取值

func	100000	100010	100011	101010	101011	（与 func 取值无关）					
op	000000	000000	000000	000000	000000	001101	001001	100011	101011	000100	000010
控制信号	add	sub	subu	slt	sltu	ori	addiu	lw	sw	beq	jump
Branch	0	0	0	0	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	0	0	0	0	1
RegDst	1	1	1	1	1	0	0	0	×	×	×
ALUSrc	0	0	0	0	0	1	1	1	1	0	×
ALUctr	add	sub	subu	slt	sltu	or	addu	addu	addu	subu	×
MemtoReg	0	0	0	0	0	0	0	1	×	×	×
RegWr	1	1	1	1	1	1	1	1	0	0	0
MemWr	0	0	0	0	0	0	0	0	1	0	0
ExtOp	×	×	×	×	×	0	1	1	1	×	×

2. 控制单元设计

在分析每条指令中控制信号取值的基础上，可以设计控制单元。从表 6.5 可看出，除

了 ALUctr 外，其他都是单值控制信号；而且，对于所有 R-型指令，除了 ALUctr 信号以外，其余控制信号的取值都相等，R-型指令的 ALUctr 信号的取值由其 func 字段决定。因此，可以考虑单独用一个局部的 ALU 控制器来对 R-型指令进行译码，译码输出为 ALUctr 信号。

如图 6.30 所示，控制器分成主控制器和局部 ALU 控制器两部分。主控制器的输入为指令操作码 op，输出各种控制信号，并根据指令所涉及的 ALU 运算类型产生 ALUop，同时，生成一个 R-型指令的控制信号 R-type，用它来控制选择将 ALUop 输出作为 ALUctr 信号，还是根据 R-型指令中的 func 字段来产生 ALUctr 信号。

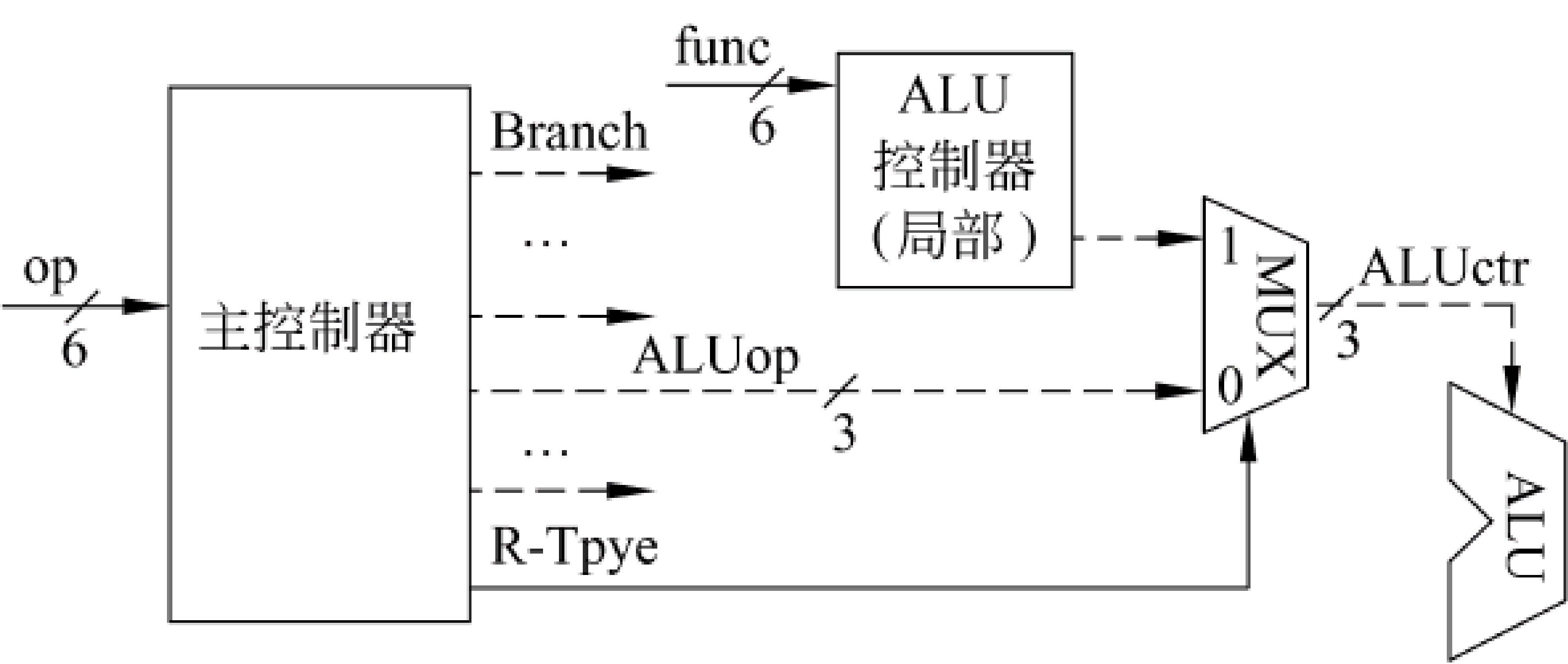


图 6.30　指令译码器的设计

对于本节给出的 11 条目标指令，ALUop 需要区分的可能情况如下。

(1) R-型指令：由局部 ALU 控制器根据 func 字段来确定 ALUctr。

(2) I-型 ori 指令：ALUop＝ALUctr＝or。

(3) I-型 addiu 指令：ALUop＝ALUctr＝addu。

(4) I-型 lw/sw 指令：ALUop＝ALUctr＝addu。

(5) I-型 beq 指令：ALUop＝ALUctr＝subu。

(6) J-型指令：ALUop＝ALUctr＝任意。

因为 ALUctr 需要对所有指令涉及的 ALU 操作类型进行编码，而 ALUop 只要对非 R-型指令涉及的 ALU 操作类型进行编码，所以，ALUop 表示的运算种类一定不会超过 ALUctr 表示的运算种类。因为本章节的设计目标仅有 11 条指令，用三位编码即可表示所有运算类型。如果要实现完整的 MIPS 指令系统，ALUop 和 ALUctr 还需要对更多的情况进行编码，因而它们可能需要更多的位数。

对 ALUop 编码时，在非 R-型指令的情况下，就取和 ALUctr 一样的编码，然后再选剩下一种未用编码对 R-型指令编码。表 6.6 就是根据表 6.3 中 ALUctr 的编码而得到的一种对 ALUop 的编码方案。

表 6.6　ALUop 的编码分配

	R-型指令	ori	addiu	lw/sw	beq	jump
ALUctr	add/sub/subu/slt/sltu/…	or	addu	addu	subu	任意
ALUop<2：0>	001(或 011、101、110、111)	010	000	000	100	xxx

（1）主控制器的设计

根据表 6.5 和表 6.6，可写出主控制器的各控制信号的逻辑表达式。假定 R-型指令对应的 ALUop 的编码为 001，操作码 op 各位分别表示为 op＜5＞ op＜4＞op＜3＞op＜2＞op＜1＞op＜0＞，则 Branch、RegWr 和 ALUop 的逻辑表达式如下。

```
Branch    =beq= !op<5>&!op<4>&!op<3>&op<2>&!op<1>&!op<0>
RegWr     =R-type+ori+addiu+lw
          = !op<5>&!op<4>&!op<3>&!op<2>& !op<1>& !op<0>     (R-type)
            + !op<5>&!op<4>&op<3>&op<2>&!op<1>&op<0>        (ori)
            + !op<5>&!op<4>&op<3>&!op<2>&!op<1>&op<0>       (addiu)
            + op<5>&!op<4>&!op<3>&!op<2>&op<1>&op<0>        (lw)
ALUop<2>=beq= !op<5>&!op<4>&!op<3>&op<2>&!op<1>&!op<0>
ALUop<1>=ori= !op<5>&!op<4>&op<3>&op<2>&!op<1>&op<0>
ALUop<0>=R-type= !op<5>&!op<4>&!op<3>&!op<2>&!op<1>&!op<0>
```

根据各控制信号的逻辑表达式，可方便地画出控制器逻辑电路。如图 6.31 所示，主控制器可用一个 PLA 电路实现，其中的“与”阵列是指令译码器。

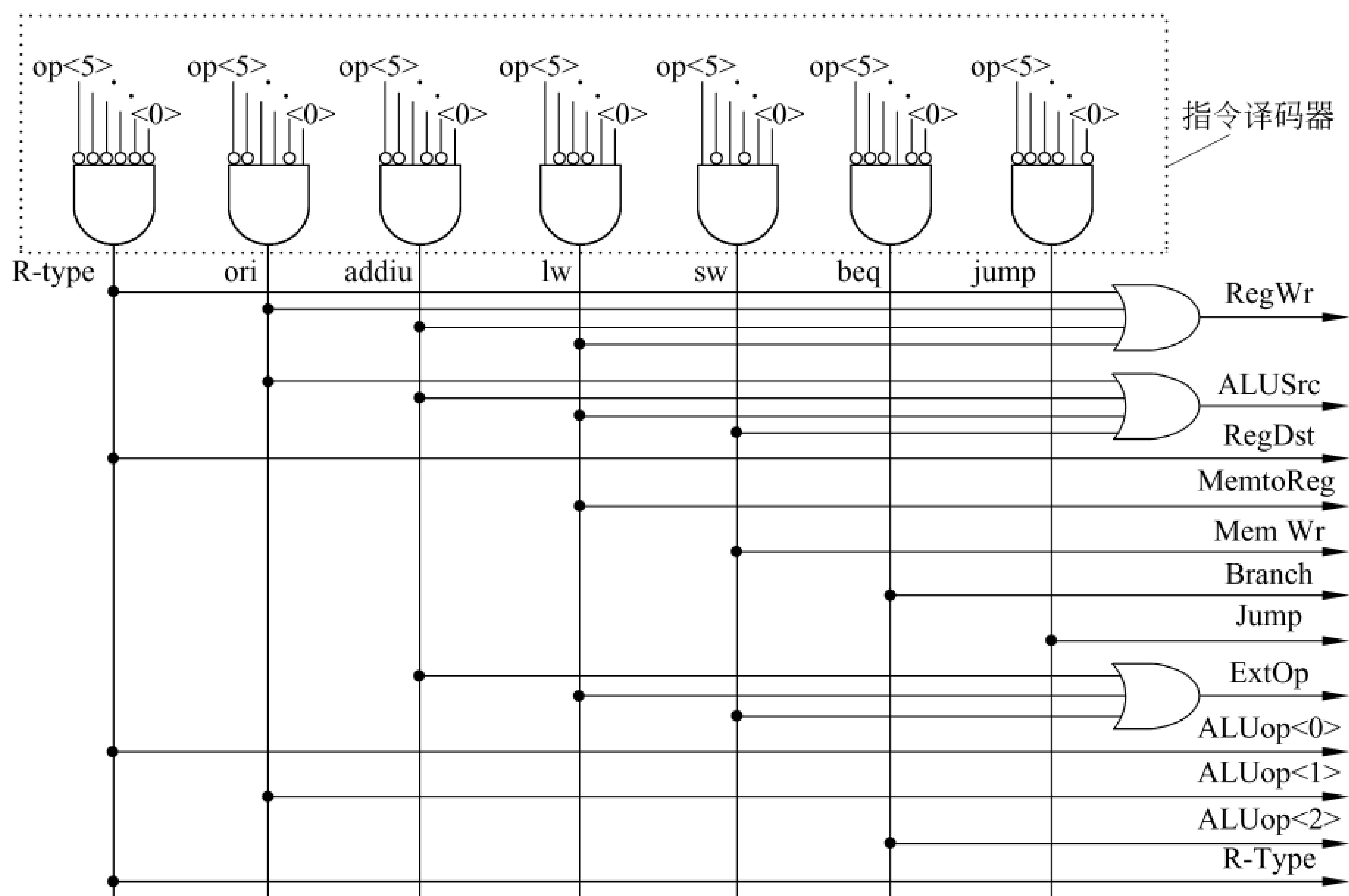


图 6.31 主控制器逻辑电路

（2）ALU 局部控制器的设计

ALU 局部控制器的输入为 func 字段，输出为 ALU 操作控制信号 ALUctr。表 6.7 给出了部分 MIPS 指令系统中 R-型指令的 func 字段编码分配情况，其中前 5 条指令是本章节所要实现的 11 条目标指令中包含的。

如果仅考虑表中所列部分，则可不考虑 func＜5：4＞，因为 func 字段前两位总是 10。根据表 6.7 中列出的 func＜3：0＞与 ALUctr 之间的关系，可以方便地写出 ALU 局部控制

器中输入和输出之间的逻辑关系如下。

```
ALUctr<2>=!func<2>&func<1>
ALUctr<1>=func<3>&!func<2>&func<1>+!func<3>&func<2>&!func<1>&func<0>
ALUctr<0>=!func<3>&!func<2>&!func<0>+!func<2>& func<1>&!func<0>
```

根据上述逻辑关系，不难画出局部 ALU 控制器的逻辑电路图。

表 6.7 中仅列出了部分 R-型指令的编码情况，如果要实现完整的 MIPS 指令，则 func 字段前 2 位也必须考虑，ALUctr 还需用更多位数才能表示所有 ALU 操作控制信号。

表 6.7 func 字段编码分配表

func<5：0>	MIPS 指令	ALU 操作类型	ALUctr<2：0>
…	…	…	…
1 0 0 0 0 0	add	add	0 0 1
1 0 0 0 1 0	sub	sub	1 0 1
1 0 0 0 1 1	subu	subu	1 0 0
1 0 1 0 1 0	slt	slt	1 1 1
1 0 1 0 1 1	sltu	sltu	1 1 0
…	…	…	…
1 0 0 0 0 1	addu	addu	0 0 0
1 0 0 1 0 1	or	or	0 1 0
1 0 0 1 0 0	and	and	…
1 0 0 1 1 0	xor	xor	…
1 0 0 1 1 1	nor	nor	…
…	…	…	…

6.2.4 时钟周期的确定

计算机性能(即程序执行速度)由三个关键因素决定：指令数目、时钟周期和 CPI。其中，指令数目由编译器和指令集决定，而时钟周期和 CPI 由处理器的设计与实现决定。因此，处理器的设计与实现非常重要，它直接影响计算机的性能。单周期处理器每条指令在一个时钟周期内完成，所以 CPI 为 1，而时钟周期往往很长，通常取最复杂指令所用的指令周期。在给出的 11 条指令中，很显然，最长的是 lw 指令周期。

图 6.32 给出了 lw 指令执行定时过程，从图中可以看出，lw 指令周期所包含的时间为 PC 锁存延迟(Clk-to-Q)＋取指令时间＋寄存器取数时间＋ALU 延迟＋存储器取数时间＋寄存器建立时间＋时钟扭斜。

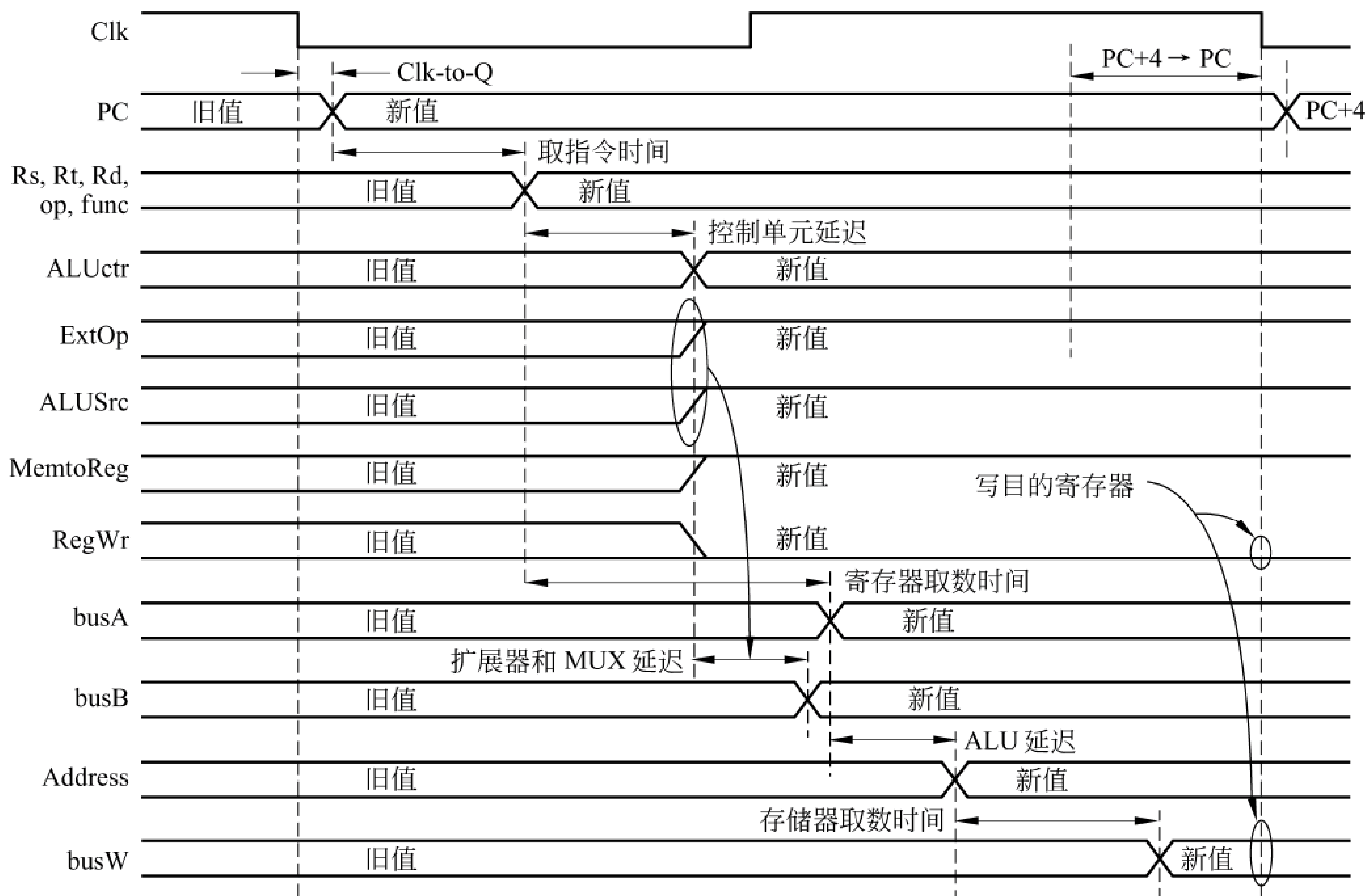


图 6.32　Load 指令执行定时

6.3　多周期处理器设计

单周期处理器时钟周期远远大于许多指令实际所需执行时间，例如，R-型指令和立即数运算指令都不需要读内存；Store 指令不需要写寄存器；分支指令不需要访问内存和写寄存器；Jump 不需要 ALU 运算，不需要读内存，也不需要读写寄存器。受时钟周期宽度的影响，单周期处理器的效率低下、性能极差，实际上，现在很少用单周期方式设计 CPU。我们介绍单周期 CPU 的设计实现，只是为了有助于理解实际的多周期执行和流水线执行两种方式。

*6.3.1　信号竞争问题

多周期处理器的基本思想为：把每条指令的执行分成多个大致相等的阶段，每个阶段在一个时钟周期内完成；各阶段内最多完成一次访存或一次寄存器读写或一次 ALU 操作；各阶段的执行结果在下个时钟到来时保存到相应存储单元或稳定地保持在组合电路中；时钟周期的宽度以最复杂阶段所用时间为准，通常取一次存储器读写的时间。

在介绍单周期处理器时，存储器被简化为理想情况，即假定每次写操作都有时钟控制，并且在每次时钟到来时，地址、数据和写使能信号都已稳定一段时间。事实上，存储器的实际写操作不是由时钟边沿触发，而是一个组合逻辑电路。其写操作的过程为：当"写使能"信号有效，并且写入数据和地址已稳定，则经过一个写操作时间后，数据被写入。这里，重要的一点是，地址和数据必须在"写使能"信号有效前先稳定在各自的输入端。实际的存储器

在单周期数据通路中不能可靠工作，这是因为：不能保证地址和数据能在“写使能”信号有效前稳定，即地址、数据和“写使能”之间存在竞争(Race)问题。

竞争问题有时会导致机器意外出错，甚至崩溃。在多周期处理器中，可通过以下方式来解决竞争问题：首先确认地址和数据在第 n 周期结束时已稳定，然后，使“写使能”信号在一个周期后(即第 $n+1$ 周期)有效，并使地址和数据在“写使能”信号无效前不改变其值。

*6.3.2 指令执行状态分析

多周期处理器中，每条指令分多个阶段执行，每个阶段占一个时钟周期，称为一个状态。因此，一条指令的执行过程由多个状态组成。在指令被译码之前，每条指令所完成的操作是一样的，指令译码后不同的指令有不同的执行过程。

如图 6.33 所示是加了控制信号的多周期数据通路示意图。因为 PC、IR、分支目标寄存器(Branch Target)(简称 BT 寄存器)和寄存器堆只能在需要时写入新值，所以它们都需有“写使能”信号来控制；而寄存器 A 和 B 是临时寄存器，每来一个时钟都可改变它们的值，通过对后续多路选择器的控制，能保证数据通路正确执行，因而，寄存器 A 和 B 无需“写使能”控制信号。

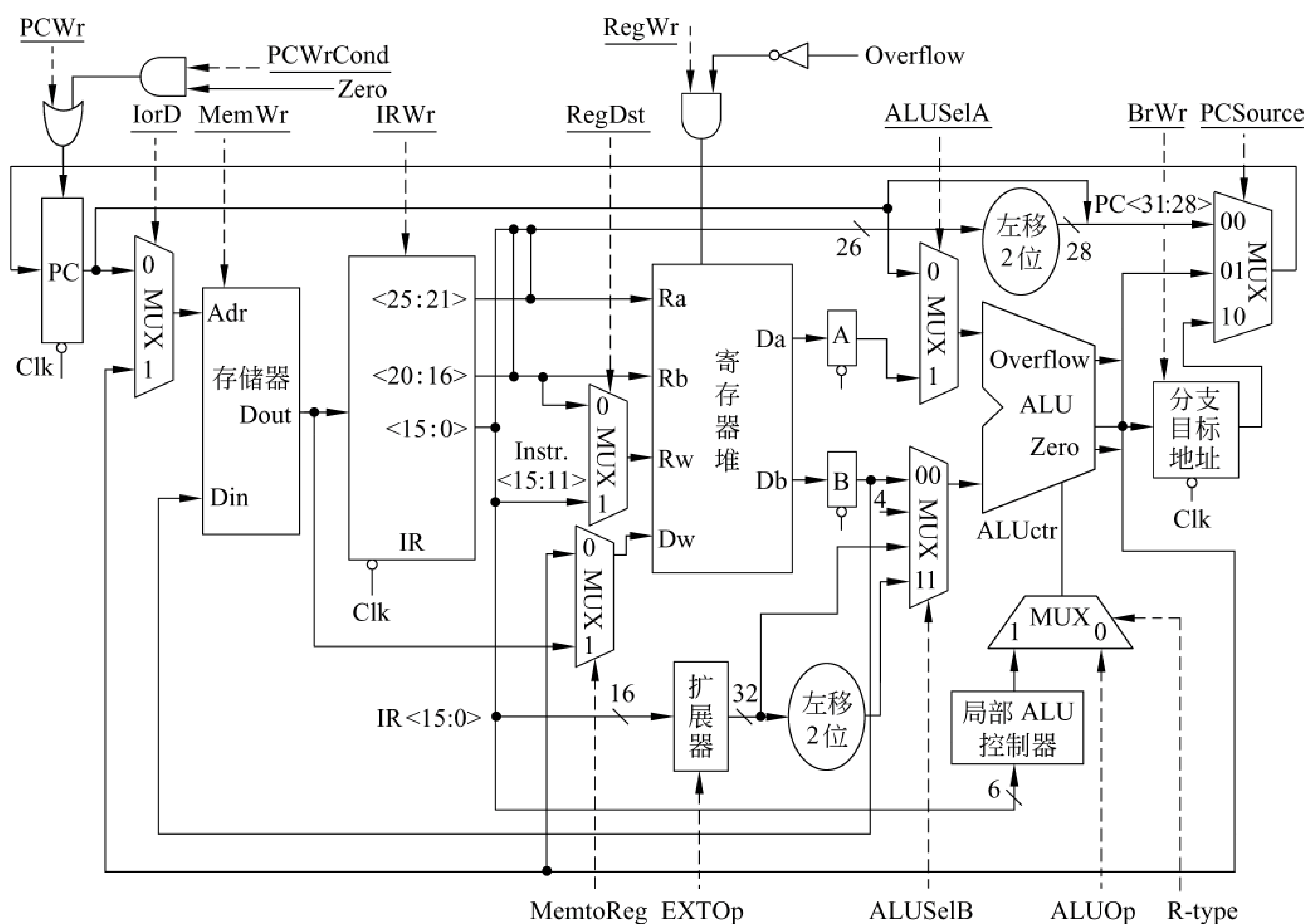


图 6.33 带控制信号的多周期数据通路

PC 的来源可以是 PC+4、分支目标地址和无条件跳转目标地址，由 PCSource 信号控制。分支目标地址保存在专门的分支目标地址寄存器中，无条件跳转目标地址由 PC 高 4 位和指令低 26 位直接拼接后在低位添“00”得到。同单周期数据通路一样，对于需要判断

溢出的算术运算类指令，当发生溢出时，禁止写结果到寄存器堆。

1. 取指令、指令译码/取数阶段

取指令、指令译码/取数是所有指令执行之前的公共操作，与指令译码结果没有关系。

(1) 取指令状态

取指令阶段的功能是 IR←M[PC]、PC←PC+4。因此，需要将 PC 的值作为地址读存储器，并将读出指令送 IR 输入端，使得下个时钟到来时，读出的指令送 IR。同时，PC 将送 ALU 的 A 口，并选择 4 送 ALU 的 B 口，控制 ALU 做不判溢出的加法操作，得到 PC+4，送 PC 输入端，使得下个时钟到来时，PC+4 送 PC。

该状态名记为 IFetch，控制信号取值为 IorD＝0，ALUSelA＝0，ALUSelB＝01，ALUOp＝addu，PCSource＝01，PCWr＝IRWr＝1，MemWr＝RegWr＝BrWr＝R-type＝0，其余任意。

(2) 译码/取数状态

译码/取数阶段的功能是 CU(译码)←IR＜31：26＞、A←R[IR＜25：21＞]、B←R[IR＜20：16＞]。该阶段 ALU 是空闲的，所以可以利用 ALU"投机"计算分支目标地址送 BT 寄存器，如果当前指令是分支指令，则可省一个时钟周期，若不是分支指令，也不会有任何影响。

分支目标地址计算方法为 PC+4+(SignExt(imm16)×4)，因为取指令阶段结束时已经把 PC+4 送 PC 输入端，所以，该阶段只要计算：PC+(SignExt(imm16)×4)。

状态名记为 RFetch/ID，控制信号值为 EXTOp＝1，ALUSelA＝0，ALUSelB＝11，ALUOp＝addu，BrWr＝1，PCWr＝PCWrCond＝IRWr＝MemWr＝RegWr＝R-type＝0，其余任意。

2. R-型指令运算阶段

R-型指令运算执行阶段功能为 R[IR＜15：11]＞]←A op B。即 A、B 内容分别送 ALU 的 A 口和 B 口，进行相应运算后，写入到寄存器堆中。对于 R-型指令，ALU 操作控制信号 ALUctr 由局部 ALU 控制器根据 func 字段产生，而主控制器生成的 ALUop 不起作用。考虑到寄存器堆写入时的"写使能"信号和写入数据、地址信号之间的竞争问题，该阶段要用两个状态来完成。第一个状态先送数据、地址，第二个状态再使"写使能"信号有效。

R-型指令执行状态(记为 RExec)的控制信号取值为 ALUSelA＝1，ALUSelB＝00，RegDst＝R-type＝1，RegWr＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝MemtoReg＝0，其余任意。

结束状态(记为 RFinish)的控制信号取值为除 RegWr＝1 外，其余同执行状态。

3. I-型指令立即数运算阶段

I-型运算指令的执行阶段功能为 R[IR＜20：16＞]←A op Ext([IR＜15：0＞])。算术运算指令(如 addiu)对立即数进行的是符号扩展，而逻辑运算指令(如 ori)进行的是零扩展。

每条 I-型运算指令与上述 R-型指令的运算阶段一样，也分执行和结束两个状态。执行状态中有两个信号(ALUop 和 EXTOp)的值会随指令不同而不同：算术运算指令的 EXTOp 为 1，逻辑运算指令的 EXTOp 为 0；ALUOp 的取值由指令的运算类型确定，例如，ori 指令的 ALUop 为 or，andi 指令的 ALUop 为 and，addiu 指令的 ALUop 为 addu，addi 指

令的 ALUop 为 add 等。其他信号取值如下：ALUSelA＝1，ALUSelB＝10，MemtoReg＝RegDst＝RegWr＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝R-type＝0，其余任意。

结束状态控制信号取值除 RegWr＝1 外，其余信号的取值同执行状态。

例如，ori 指令执行状态（记为 oriExec）的控制信号取值为 ALUSelA＝1，ALUSelB＝10，ALUop＝or，EXTOp＝MemtoReg＝RegDst＝RegWr＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝R-type＝0，其余任意；结束状态（记为 oriFinish）的各控制信号的取值为 RegWr＝1，ALUSelA＝1，ALUSelB＝10，ALUop＝or，EXTOp＝MemtoReg＝RegDst＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝R-type＝0，其余任意。

4. lw 指令执行阶段

lw 指令执行阶段功能为 R[IR＜20：16＞]←M[A＋ SignExt([IR＜15：0＞])]。由以下三个状态组成。

访存地址计算状态（记为 MemAdr）：ALUSelA＝1，ALUSelB＝10，ALUOp＝addu，EXTOp＝IorD＝1，RegWr＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝R-type＝0，其余任意。

存储器取数状态（记为 MemFetch）：EXTOp 、ALUSelA、ALUSelB、ALUOp、IorD、R-type 和上一个状态一样，以继续保持访存地址信号的稳定；MemtoReg＝1，使数据尽早稳定在寄存器堆的 Dw 输入端；RegDst＝0，使地址尽早稳定在寄存器堆的 Rw 输入端；控制所有寄存器和存储器不做任何更新，即 RegWr＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝0；其余任意。

结果写回寄存器状态（记为 lwFinish）：RegWr＝1，使数据写入寄存器，其余信号取值同上一个状态，以继续保持寄存器堆的 Dw 和 Rw 输入端稳定不变。

5. sw 指令执行阶段

sw 指令执行阶段功能为 M[A＋SignExt([IR＜15：0＞])]←B。由访存地址计算和存储器存数（记为 swFinish）两个状态组成。第一个状态同 lw 指令第一个状态，第二个状态只要使 MemWr＝1，其余控制信号不变，这样保证存储器的写入地址和写入数据在本状态中稳定不变。

6. 分支指令执行阶段

分支指令 beq 执行阶段的功能为 if (A－B＝0) then PC←分支目标地址。因此只要一个状态即可，该状态名记为 BrFinish。其控制信号取值为 ALUSelA＝1，ALUSelB＝00，ALUOp＝subu，PCWrCond＝1，PCSource＝10，RegWr＝PCWr＝IRWr＝MemWr＝BrWr＝R-type＝0，其余任意。

7. 无条件跳转指令执行阶段

跳转指令 jump 执行阶段的功能为 PC←Target。因此只要一个状态即可，该状态名记为 JumpFinish。其控制信号取值为 PCSource＝00，PCWr＝1，RegWr＝ IRWr＝MemWr＝BrWr＝0，其余任意。

根据上述对每条指令执行过程的分析，得到一个状态转换图。图 6.34 是一个支持 R-型指令、I-型运算类指令 ori 以及 lw/sw、beq 和 jump 指令执行的状态转换示意图，图中每个状态用一个状态号和状态名标识，例如，“0：IFetch”表示第 0 状态，执行取指令（IFetch）操作，圆圈中示意性地给出了该状态下部分控制信号相应的取值。

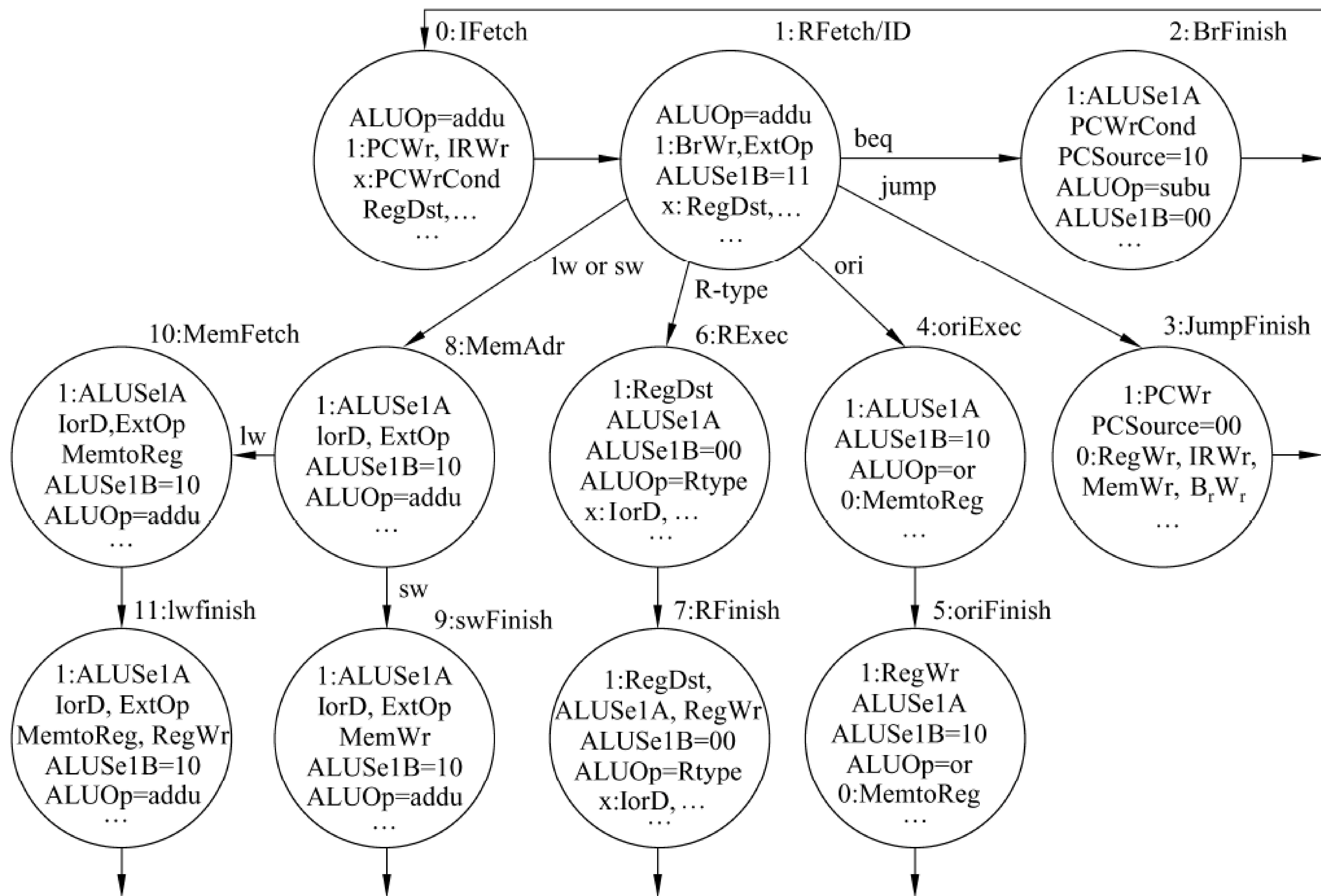


图 6.34　指令执行状态转换图

指令在图 6.33 所示的多周期数据通路中执行的过程就是图 6.34 所示的状态转换过程，每来一个时钟，进入下一个状态。从图 6.34 可看出，R-型指令、I-型运算类指令和 sw 指令的 CPI 都一样，其 CPI＝4；lw 指令的 CPI 最大，其 CPI＝5；分支指令和跳转指令的 CPI＝3。如果不在译码/取数阶段“投机”计算分支目标地址，则分支指令的 CPI 为 4。

如果需要支持更多 I-型运算类指令，可以像 ori 指令一样，每条指令增加两个状态，因此，采用这种方式得到的状态数会很多；为了减少状态数，也可以将 I-型运算类指令像 lw/sw 指令一样，先分成算术运算类和逻辑运算类两路，然后每路再像 R-型指令一样，通过综合方式合并成执行和完成两个状态，这样，I-型运算类指令一共有 4 个状态：逻辑运算执行 I-LExec、逻辑运算完成 I-LFinish；算术运算执行 I-AExec、算术运算完成 I-AFinish。这 4 个状态中的 ALUop 信号都由专门的局部 I-型运算控制器产生。该局部控制器的输入为操作码 op，输出为 ALUop。

*6.3.3　硬连线路控制器设计

在单周期控制器的实现中，由于控制信号在整个指令执行过程中不变，所以，用真值表能反映指令和控制信号之间的关系，根据真值表就能实现控制器。那么，多周期控制器能不能这样做呢？由于多周期数据通路中每个指令的执行有多个周期，每个周期的控制信号取值不同，所以，不能用简单的真值表描述的设计方式。多周期控制器通常采用基于有限状态机描述和微程序描述两种方式来实现。

有限状态机描述方式实现的控制器称为有限状态机控制器，其基本思想为：用一个有限状态机描述指令执行过程，由当前状态和操作码确定下一状态，每来一个时钟发生一次状

态改变，不同状态输出不同的控制信号值，然后送到数据通路来控制指令的执行。

图 6.35 描述了采用这种方式实现的控制器结构，它由两部分组成：一个组合逻辑控制单元和一个状态寄存器。通常用 PLA 电路实现组合逻辑控制单元。所以，这种控制器也称为组合逻辑控制器，或 PLA 控制器，或硬连线路控制器。

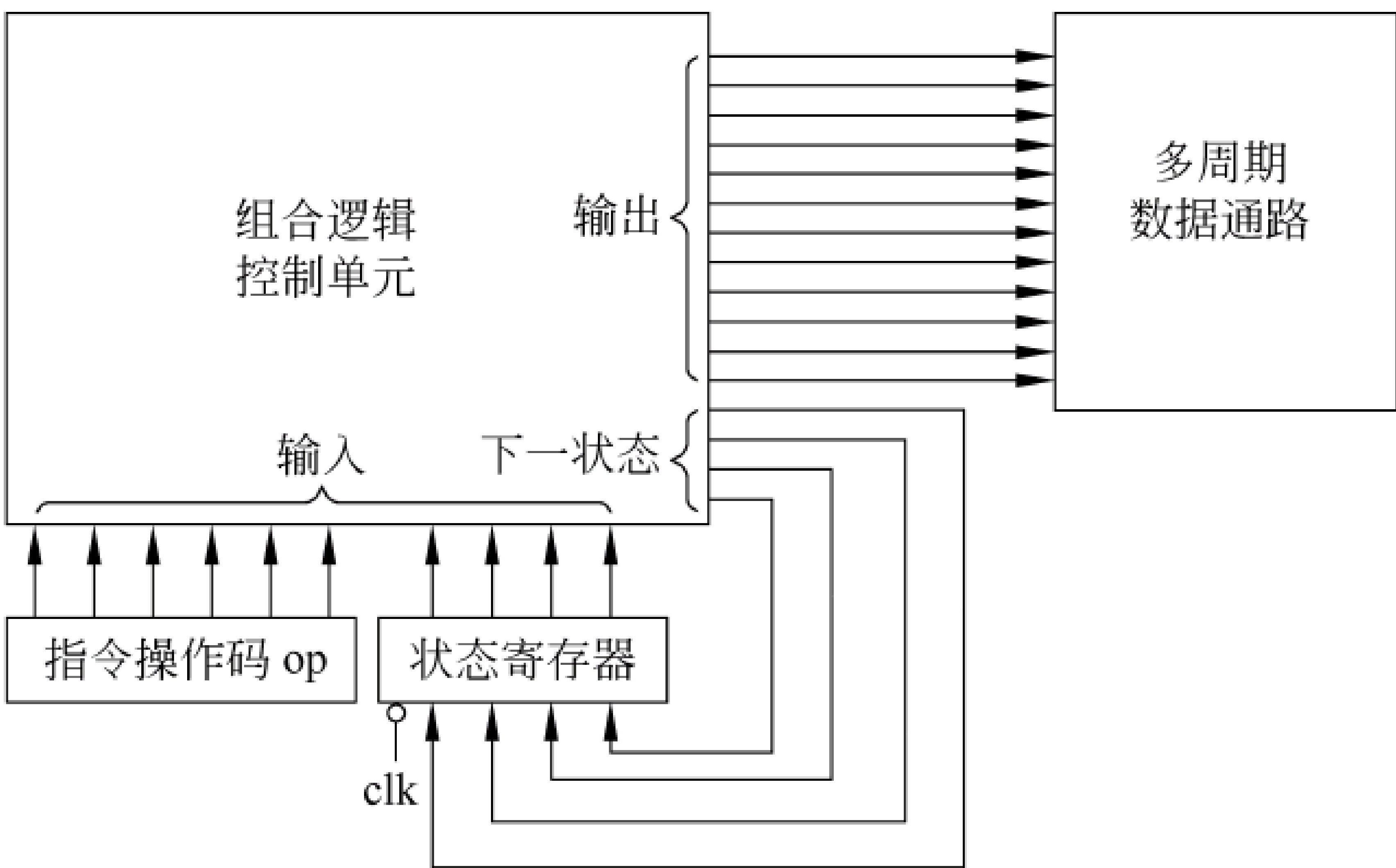


图 6.35　有限状态机控制器结构

对于图 6.35 所示的有限状态机，假定每个状态号如图 6.34 中所设，分别为 0～11，共 12 个状态，因此，状态变量要用 4 位，设分别为 S3S2S1S0。考察每个状态前面的状态和指令操作码，得到状态转换表 6.8。

表 6.8　多周期控制器状态转换表

当前状态 S3S2S1S0	指令操作码 op5op4op3op2op1op0	下一状态 NS3NS2NS1NS0
State2、3、5、7、9、11		0　0　0　0
State0		0　0　0　1
State1	000100 (beq)	0　0　1　0
State1	000010 (jump)	0　0　1　1
State1	001101 (ori)	0　1　0　0
State4		0　1　0　1
State1	000000 (R-type)	0　1　1　0
State6		0　1　1　1
State1	100011 (lw)	1　0　0　0
State1	101011 (sw)	1　0　0　0
State8	101011 (sw)	1　0　0　1
State8	100011 (lw)	1　0　1　0
State10		1　0　1　1

根据表 6.8 可画出用 PLA 电路实现的状态转换电路以及控制信号生成电路，从而实现组合逻辑控制单元，如图 6.36 所示。

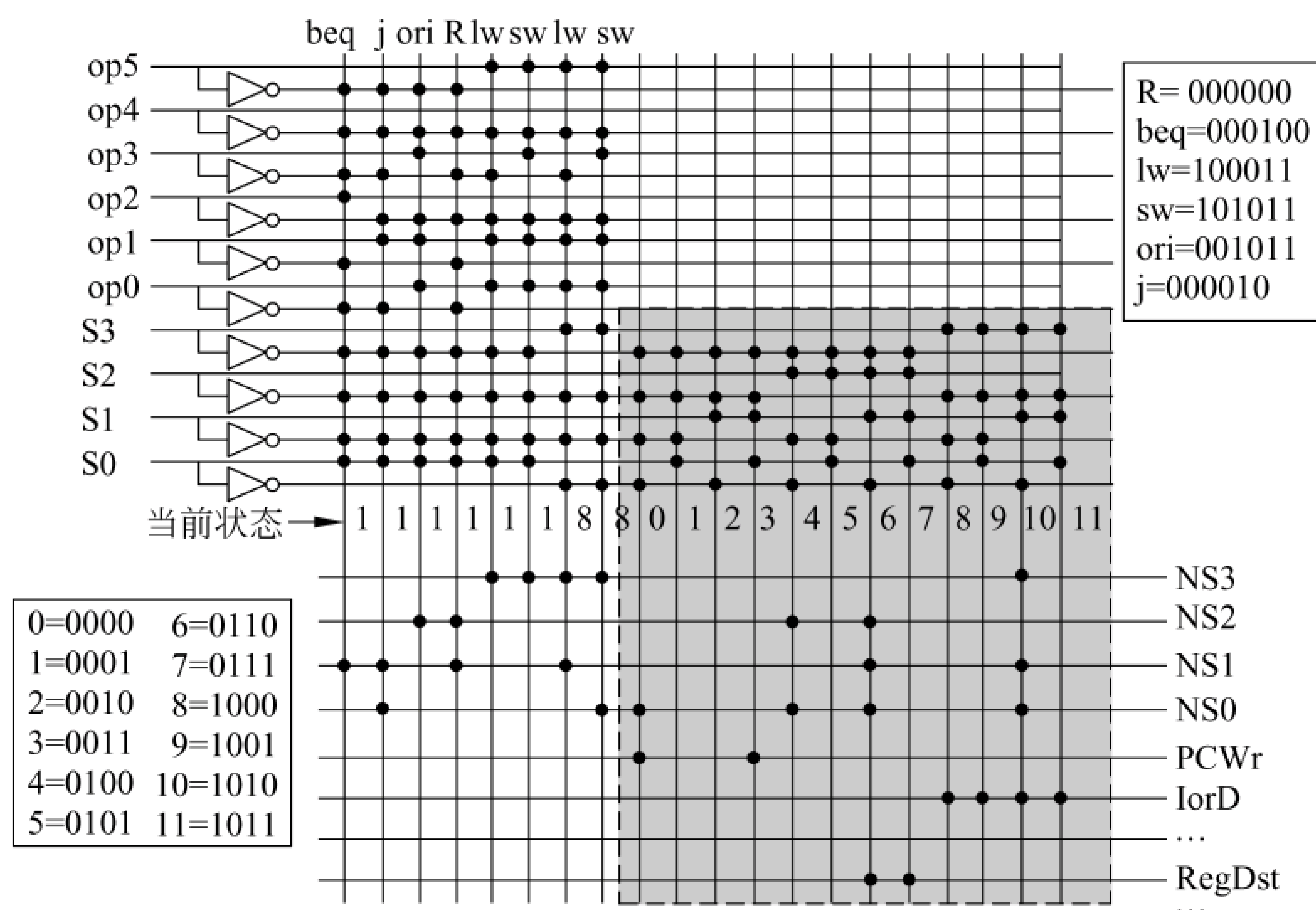


图 6.36 用 PLA 电路实现的组合逻辑控制单元

图 6.36 所示的有限状态机称为“摩尔机”，其特点是控制信号的输出仅依赖于当前状态。因此，“摩尔机”方式实现的组合逻辑控制单元被分为两部分：由操作码和当前状态确定下一状态的电路部分和由当前状态确定控制信号的电路部分(图 6.36 中由右下角虚线区域标出部分)。

例 6.1 假定多周期处理器采用图 6.33 所示的数据通路和图 6.36 所示的控制单元。若程序中各类指令所占比例为 Load：22%；Store：11%；R-型和 I-型运算：49%；Branch：16%；Jump：2%，则多周期处理器比单周期处理器大约快多少倍？

解：根据图 6.34 可知，图 6.33 所示的多周期处理器的各指令所需时钟周期数如下。Load：5；Store：4；R-型和 I-型运算：4；Branch：3；Jump：3。根据以下 CPI 计算公式，可计算出多周期处理器的 CPI。

$$\text{CPI} = \text{CPU 时钟周期数} / \text{指令数} = \sum(\text{指令数}\ i \times \text{CPI}i) / \text{指令数}$$

$$= \sum(\text{指令数}\ i / \text{指令数}) \times \text{CPI}i$$

$$\text{CPI}=0.22\times5+0.11\times4+0.49\times4+0.16\times3+0.02\times3=4.04$$

单周期 CPU 的 CPI 为 1，但时钟周期为最长的 Load 指令执行时间，约为多周期时钟宽度的 5 倍。假设单周期时钟宽度为 1，则多周期时钟周期约为单周期的 1/5，所以，多周期的总体时间约为 4.04×1/5=0.81；而单周期总体时间为 1×1=1，因此，多周期处理器的指令执行速度大约是单周期处理器的 1/0.81=1.23 倍。

6.4 微程序控制器设计

硬连线路控制器的速度快，适合于简单或规整的指令系统，例如 MIPS 指令集。但是，由于它是一个多输入/多输出的巨大逻辑网络，对于复杂指令系统来说，对应的硬连线路控

制器结构庞杂,实现困难,维护不易,扩充和修改指令相当困难。如果指令系统太复杂的话,甚至无法用有限状态机描述。所以,对于复杂指令系统或其中的复杂指令,大多采用微程序方式来设计控制器。

用微程序方式实现的控制器称为微程序控制器,其基本思想为:仿照程序设计方法,将每条指令的执行过程用一个微程序来表示,每个微程序由若干条微指令组成,每条微指令相当于有限状态机中的一个状态。所有指令对应的微程序都存放在一个 ROM 中,这个 ROM 称为控制存储器(Control Storage),简称控存(CS)。

在微程序控制器控制下执行指令时,将每条指令对应的微程序从控存中取出,在时钟的控制下,按照一定的顺序执行微程序中的每条微指令。通常一个时钟周期执行一条微指令。

*6.4.1 Wilkes 微程序控制器

1951 年,M. V. Wilkes 最先提出微程序这个概念。Wilkes 提出的设计方案如图 6.37 所示。核心部分是连接有二极管的一个阵列,每来一个时钟,阵列的一行被激活。阵列中出现二极管的地方(图中以圆点表示)产生信号,左半部产生控制信号,右半部产生下个周期将要激活的行地址,阵列的每一行对应一条微指令。当某条微指令需要实现条件转移时,用条件码触发器控制产生不同的微转移地址。每条指令微程序首地址的产生由指令寄存器 IR 中指令操作码确定。为避免译码器出现不稳定状态而采用了双地址寄存器。

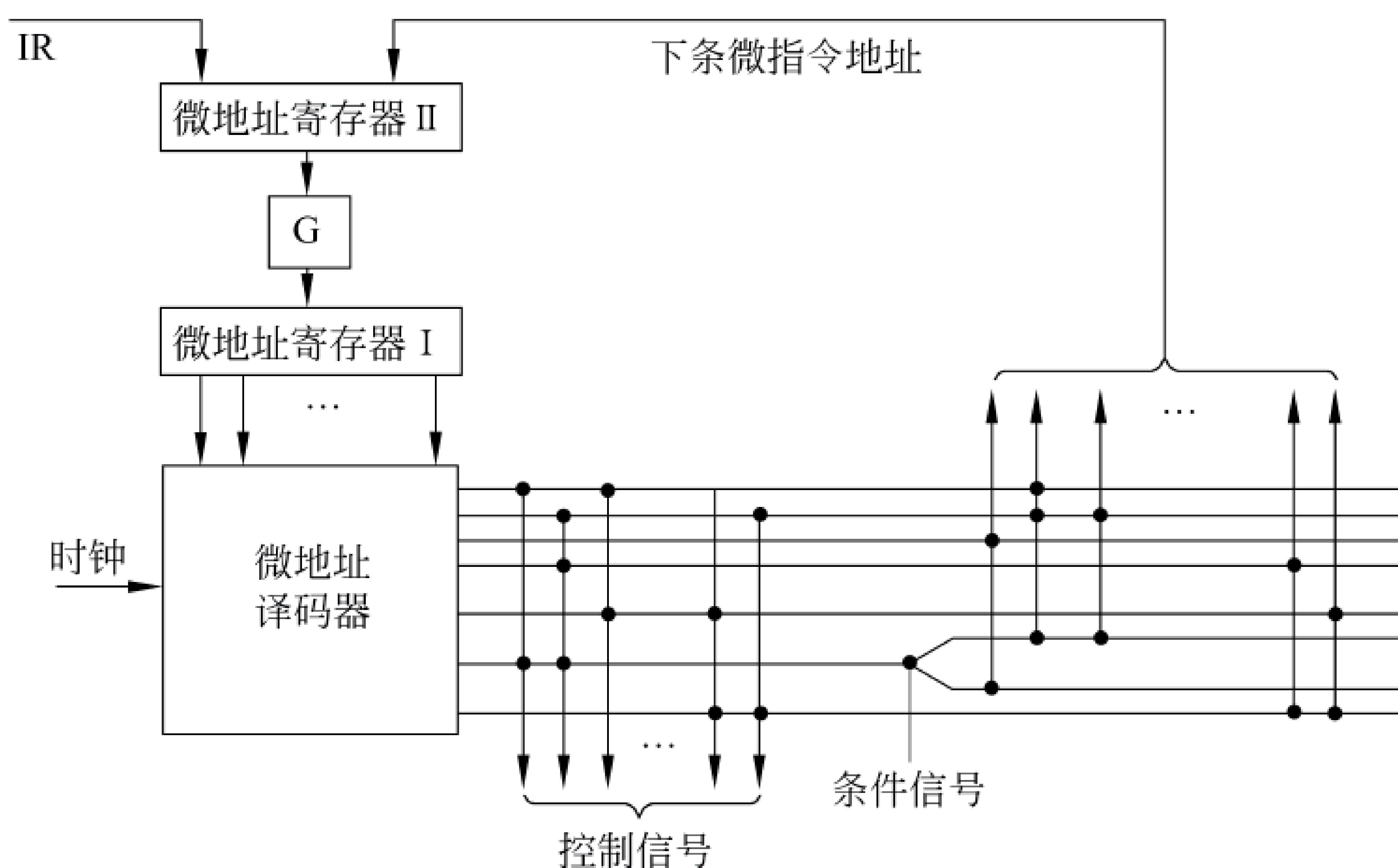


图 6.37 Wilkes 微程序控制器

微程序设计的思想给计算机控制部件的设计和实现技术带来了巨大的影响。与硬连线路设计相比,它大大降低了控制器设计的复杂性,提高了设计的标准化程度。由于机器指令的执行过程用微程序控制,因而提供了很大的灵活性,使得设计的变更、修改以及指令系统的扩充都成为不太困难的事情。它与传统的软件设计有许多类似之处,但是,由于微程序相对固定,且通常不放在主存内,故有可能利用工作速度较高的 ROM 存放微程序,从而缩短微程序的运行时间。这是一种固化了的微程序,称为固件(Firmware)。

微程序控制器的主要缺点是:比相同或相近指令系统的硬布线控制器慢。因此,RISC 机大都采用硬连线路控制器,而 IA-32 则采用了硬连线和微程序相结合的方式来实现控制逻辑。

6.4.2 微程序控制器的结构

1. 基本术语

一条指令的功能通过执行一系列基本操作来完成。这些基本操作称为微操作，每个微操作在相应控制信号的控制下执行，这些控制信号在微程序设计中称为微命令。例如，前面提到的控制信号 PC_{in}(或 PCWr)就是一个微命令，可以控制将一个地址写入 PC。

微程序是一个微指令序列，对应于一条机器指令的功能。每条微指令是一个 0/1 序列，其中包含若干个微命令，它完成一个基本运算或传送功能。有时也将微指令字称作控制字 CW (Control Word)。

2. 微程序控制器的基本结构

图 6.38 给出了微程序控制器的基本结构。其输入是指令、条件码，输出是微命令。图 6.38 中使用了一个微程序计数器 μPC(microprogram counter)，用来指出微指令在控存中的地址。每次把新指令装入 IR 时，“起始和转移地址发生器”将根据指令的内容，生成微程序的入口地址放入 μPC 中，以后每来一个时钟，μPC 自动增值(+“1”)，这样，依次从控存中读出一条条微指令执行。μIR 为微指令寄存器，存放从控存取出的微指令，每条微指令被译码后，产生一系列微命令，送到数据通路中。机器指令的执行过程常常与条件码(即状态标志)有关，因此微程序中也引入了条件转移概念。微指令中的“转移控制”部分，被送到转移地址发生器，根据条件码及相应微命令产生新的微指令地址送入 μPC。

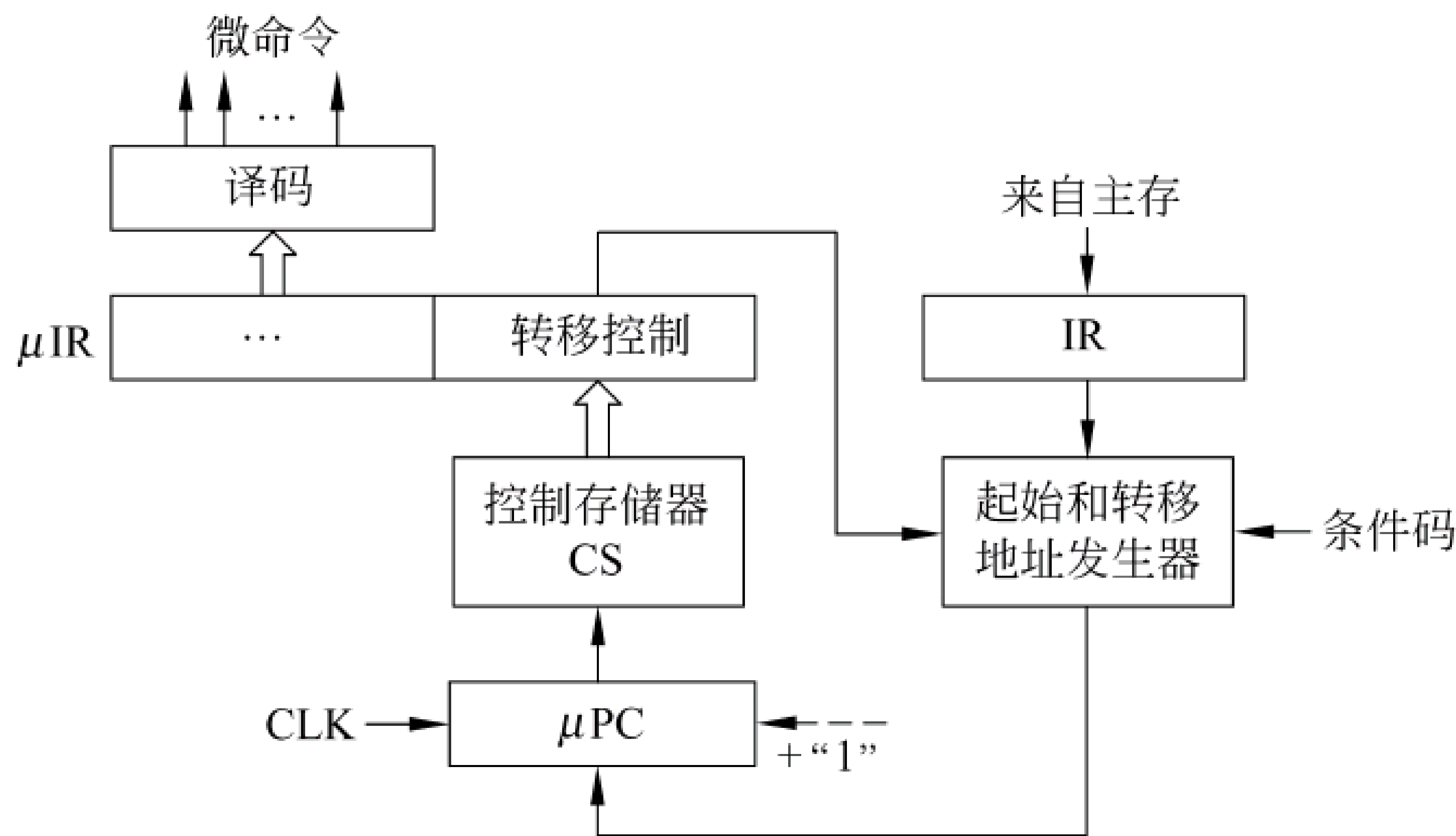


图 6.38 微程序控制器基本结构

CPU 执行程序的过程就是不断地取指令、执行指令的过程。取指令过程是每条指令的公共操作，可以专门用一个取指令微程序来实现。因此，微程序控制器的工作流程就是不断地执行取指令微程序和执行相应指令功能微程序的过程。

程序由指令组成，程序的执行要解决以下两个问题：(1)指令格式和操作码编码问题；(2)下条指令地址确定问题。微程序由微指令组成，微程序的执行也要解决类似的两个问题：(1)微指令格式和微命令编码问题；(2)下条微指令地址确定问题。

为了加快指令执行速度，通常采用定长微指令字格式，每条微指令也有两部分组成：微操作码部分(μOP)和微地址码部分(μAddr)。

6.4.3 微命令编码和微指令格式

微指令字中微操作码部分的格式设计，主要由微命令编码方式决定。微命令编码方式有以下 4 种：不译法（直接控制法）、字段直接编码（译）法、字段间接编码（译）法、最小（最短、垂直）编码（译）法。

1. 直接控制法

一位对应一个微命令，不进行微命令的编码，所以，微操作码的长度与所有微命令的个数相当，无须译码，因此，也称为不译法。对于二值微命令，本来就只占一位，没有增加位数；对于多值微命令，因为没有进行编码，所以相对来说增加了位数。例如对于 4-1 多路选择器，若编码的话只需两位，而不编码的话则需 4 位，相当于 4 个微命令；又如，对于 ALU 操作微命令 ALUctr，若 ALU 有 16 种操作，相当于 16 个微命令，编码的话只需 4 位，不编码的话则要 16 位。

显然，直接控制法的并行控制能力强，不必译码，控制电路简单、速度快。但是，由于一台机器的微命令个数很多，通常达几百个，因此微指令字可能多达几百位，实现起来非常困难。此外，因为一条微指令中往往只包含很少几个微操作，所以只有很少几位对应的微命令为 1，使得几百位的微指令字中只有少数几位为 1，因而编码空间利用率低。

2. 字段直接编码法

数据通路中的微操作之间存在两种关系：相容和互斥。相容微操作是指在数据通路中能同时进行的微操作，对应的微命令称为相容微命令；互斥微操作是指不能同时进行的微操作，对应的微命令称为互斥微命令。

字段直接编码法的基本思想是：将微指令分成若干字段，每个字段包含若干微命令。把相斥微操作组合在同一字段，相容微操作组合在不同字段，编码时对每一字段内的微操作进行。因此，通常一条微指令中最多可同时发出的微操作数就是微命令字段的个数。

例 6.2 对于图 6.9 和图 6.10 所示的单总线数据通路，假定有 4 个通用寄存器 R0，R1，R2 和 R3，ALU 操作类型共 16 种，内存和 CPU 之间采用“异步”方式通信，存取操作有 Read 和 Write 两种信号控制。每条指令执行结束时，都要执行一个公共操作，用来进行指令结束处理（如查询是否有外部“中断”请求），指令结束处理由控制信号 End 控制。要求：分别写出采用直接控制法和字段直接编码法的微操作码格式。

解：在图 6.9 所示的单总线数据通路中没有标出控制信号（即微命令），但不难看出其控制信号包括以下几类。

（1）根据图 6.10 所示，控制在寄存器和总线之间进行传送的控制信号有 17 个：$R0_{in}$、$R0_{out}$、$R1_{in}$、$R1_{out}$、$R2_{in}$、$R2_{out}$、$R3_{in}$、$R3_{out}$、Y_{in}、Z_{out}、MAR_{in}、MAR_{out}、MDR_{in}、MDR_{out}、PC_{in}、PC_{out}、IR_{in}。

（2）ALU 操作类型有 16 种：add/sub/or/and/xor/…/mov。

（3）ALU 进位控制信号有一个：$1 \to C_0$。

（4）暂存器 Y 清 0 控制信号有一个：ClearY。

（5）内存读写控制信号有三个：Read、Write、WMFC。

（6）结束控制信号有一个：End。

如果采用直接控制法，则微指令字中微操作码部分的长度就是控制信号的总个数。本

例中共有控制信号(微命令)数为 17+4+1+1+3+1=27 个,因此,微操作码共 27 位,若某位为 1,则对应的微命令有效,否则,对应微命令无效。需要说明的是,对于 ALU 操作控制信号,其控制信号的个数并不是 16,而是 4,这个是由 ALU 的结构和功能确定的(参见第 3 章 3.2.3 节)。

如果采用字段直接编码法,则需根据微操作之间的相容和互斥关系进行分组。假设 ALU 操作控制信号为 ALUop＜3：0＞,则分组和编码情况如表 6.9 所示。

表 6.9　单总线数据通路微命令分组情况及其编码表

组　名	微操作(编码)	微命令取值	说　明
寄存器送总线控制 (R_{out})	No action(0000) R0→Bus(0001) R1→Bus(0010) R2→Bus(0011) R3→Bus(0100) Z→Bus(0101) MAR→Bus(0110) MDR→Bus(0111) PC→Bus(1000)	所有都为 0 $R0_{out}$=1,其余为 0 $R1_{out}$=1,其余为 0 $R2_{out}$=1,其余为 0 $R3_{out}$=1,其余为 0 Z_{out}=1,其余为 0 MAR_{out}=1,其余为 0 MDR_{out}=1,其余为 0 PC_{out}=1,其余为 0	(1) 有些微指令中,不需有寄存器内容打到总线上,故“No action” (2) 某一时刻只能有一个寄存器内容可以打到总线,此时,相应寄存器的输出微命令取值为 1,其余寄存器的输出微命令取值为 0
寄存器输入控制 (R_{in})	No action(000) Bus→R0(001) Bus→R1(010) Bus→R2(011) Bus→R3(100) Bus→Y(101) Bus→PC(110) Bus→IR(111)	所有都为 0 $R0_{in}$=1,其余为 0 $R1_{in}$=1,其余为 0 $R2_{in}$=1,其余为 0 $R3_{in}$=1,其余为 0 Y_{in}=1,其余为 0 PC_{in}=1,其余为 0 IR_{in}=1,其余为 0	某一时刻,可以同时将总线内容送多个寄存器,所以,组内这些微操作并不互斥,但基本上不会同时发生,为了缩短微操作码的长度,将它们分在同一组
地址、数据寄存器输入控制(MR_{in})	No action(00) MAR_{in}(01) MDR_{in}(10)	所有都为 0 MAR_{in}=1,MDR_{in}=0 MAR_{in}=0,MDR_{in}=1	可能和 R_{out} 组操作同在一个节拍内发生,故分在不同组
ALU 操作控制 (ALUop)	add(0000) sub(0001) and(0010) or(0011) xor(0100) … mov(1111)	ALUop=0000 ALUop=0001 ALUop=0010 ALUop=0011 ALUop=0100 … ALUop=1111	ALU 的操作由 ALU 的操作控制端 ALUop 控制
ALU 进位控制 (C_{in})	0→C_{in}(0) 1→C_{in}(1)	1→C_0=0 1→C_0=1	
暂存器清 0 控制 (ClearY)	No action(0) ClearY(1)	ClearY=0 ClearY=1	
内存读写控制 (MEMop)	No action(00) Read(01) Write(10)	Read=0,Write=0 Read=1,Write=0 Read=0,Write=1	(只有一个读写控制信号也行,参照第 4 章 4.4.2 节)
等待 MFC 控制 (WMFC)	No action(0) 采样 MFC 信号(1)	WMFC=0 WMFC=1	WMFC=1,则启动 CPU 对 MFC 信号进行采样,若有效,则说明内存完成读写,否则,CPU 继续等待
指令结束控制 (END)	No action(0) 启动中断查询等(1)	End=0 End=1	End=1,则启动 CPU 对中断请求信号采样,若有效,则进入中断响应,否则,CPU 继续执行下条指令

从表6.9可看出,采用字段直接编码方式,共有9组,其微操作码的位数从直接控制方式的27位减少到了4+3+2+4+1+1+2+1+1=19位。

例6.3 对于图6.33所示的多周期数据通路,假定采用字段直接编码法,则微指令字可划分为几个微命令字段?请给出一种微命令编码方案,并根据该方案写出微程序。

解:图6.33所示的多周期数据通路中共有15种控制信号(共19位):EXTOp,ALUSelA,ALUSelB,ALUOp,IorD,RegWr,PCWr,PCWrCond,IRWr,MemWr,BrWr,MemtoReg,PCSource,RegDst,R-type。涉及的微操作分为9组,每组内微操作互斥,分组情况以及其中的一种编码方案如表6.10所示。

表6.10 多周期数据通路微操作分组情况及其编码表

组　名	微操作(编码)	微命令取值	说　明
ALU操作控制(ALUOp) 注:参见表6.6的编码分配	addu(000)	ALUOp=000	lw/sw指令计算访存地址或addiu执行时,进行addu操作
	subu(100)	ALUOp=100	beq指令比较大小时,进行subu操作
	or(010)	ALUOp=010	ori指令时ALU进行or操作
	R-Type(001)	ALUOp=001	由func确定ALU操作类型
ALU的A端口输入控制(ALUSelA)	PC→A口(0)	ALUSelA=0	PC作为ALU第一个操作数
	A→A口(1)	ALUSelA=1	A作为ALU第一个操作数
ALU的B端口输入控制(ALUSelB)	B→B口(00)	ALUSelB=00	B作为ALU第二个操作数
	4→B口(01)	ALUSelB=01	4作为ALU第二个操作数
	Ext→B口(10)	ALUSelB=10	扩展器输出作为ALU第二个操作数
	Ext<<2→B口(11)	ALUSelB=11	扩展器内容乘4作为ALU第二个操作数
寄存器操作控制(RegOp)	ori-Ready(000)	RegWr=0,RegDst=0,MemtoReg=0	ori指令结果写寄存器堆前先送数据和地址
	lw-Ready(001)	RegWr=0,RegDst=0,MemtoReg=1	lw指令结果写寄存器堆前先送数据和地址
	R-Ready(010)	RegWr=0,RegDst=1,MemtoReg=0	R-型指令结果写寄存器堆前先送数据和地址
	Read(011)	RegWr=0,RegDst=×,MemtoReg=×	将IR<25:21>、IR<20:16>分别作为寄存器读地址,读出后分别送A和B
	Write ALU-R(100)	RegWr=1,RegDst=1,MemtoReg=0	R-型指令结果写寄存器堆
	Write ALU-ori(101)	RegWr=1,RegDst=0,MemtoReg=0	ori指令结果写寄存器堆
	Write D_{out}(110)	RegWr=1,RegDst=0,MemtoReg=1	lw指令取出内存单元内容写寄存器堆

续表

组　名	微操作(编码)	微命令取值	说　明
内存访问控制(MemOp)	Read Instruction (00)	MemWr=0,IorD=0,IRWr=1	PC为地址,读出指令送IR
	Read Data(01)	MemWr=0,IorD=1,IRWr=0	ALU结果作为地址,读内存
	Write Data(10)	MemWr=1,IorD=1,IRWr=0	ALU结果作为地址,写内存
扩展器操作控制(ExtOp)	SignExt(1)	ExtOp=1	计算访存地址或分支目标地址时需要进行符号扩展
	ZeroExt(0)	ExtOp=0	ori指令需要进行零扩展
R-型指令ALU操作控制(RType)	R-type(1)	R-type=1	由func确定ALU操作类型
	NR-type(0)	R-type=0	由op确定ALU操作类型
分支目标地址写入控制(BrWr)	Keep Branch-Target the same(0)	BrWr=0	在分支目标地址读出之前,不能改变BT寄存器的值
	Write Branch-Target (1)	BrWr=1	事先计算出的分支目标地址需要保存到BT寄存器中,以便在需要时读出送PC
写PC控制(PCWrOp)	Keep PC the same (00)	PCSource=××, PCWr=0,PCWrCond=0	在译码/取数、非转移指令执行等阶段不能改变PC的值
	PC+4(01)	PCSource=01, PCWr=1, PCWrCond=×	顺序执行下条指令
	PC+4+Ext<<2 (10)	PCSource=10, PCWr=0, PCWrCond=1	分支条件满足时,指令转移到分支目标地址执行
	PC[31-28] ‖ IR[25-0] ‖ 00(11)	PCSource=00, PCWr=1, PCWrCond=×	无条件转移目标地址送PC

表6.10中有6组是单个微命令控制一个微操作的情况,所以,这些字段的微命令编码很简单,就用微命令的取值作为编码;还有三组是多个微命令控制一种微操作,所以,需要进行编(译)码,编码后可缩短微命令字段长度。例如,内存访问控制字段(MemOp)从三位缩短到两位,写PC控制字段(PCWrOp)从4位缩短到两位。但是,寄存器操作控制字段(RegOp)包含的微操作有7个,需要用三位编码,而本身包含的微命令也只有三个,不编码的话也只要三位,而且执行时不需要译码,所以,这个字段还是不组合为好。按表6.10表示的微指令字(仅包含微命令字段)的长度为16位,实际上也仅比未编码的直接控制方式少三位,并没有节省多少空间。因此,到底采用直接控制法还是采用编码法,需要很好地权衡。

每种微指令字组合相当于上述图6.34中的一个状态,可将图6.34的有限状态机用微程序来表示,根据表6.10给出的编码方案得到的微指令结果如表6.11所示。表中×表示任意,可以用0或1代替。这样每个状态下的0/1序列就构成了一条微指令,共有12条微指令。

表 6.11　图 6.34 中有限状态机的各状态对应的微指令

状态号	ALUOp	ALUSelA	ALUSelB	RegOp			MemOp	ExtOp	Rtype	BrWr	PCWrOp
				RegWr	RegDst	MemtoReg					
0	000	0	01	0	×	×	00	×	0	0	01
1	000	0	11	0	×	×	01	1	0	1	00
2	100	1	00	0	×	×	01	×	0	0	10
3	×××	×	××	0	×	×	01	×	0	0	11
4	010	1	10	0	0	0	01	0	0	0	00
5	010	1	10	1	0	0	01	0	0	0	00
6	001	1	00	0	1	0	01	×	1	0	00
7	001	1	00	1	1	0	01	×	1	0	00
8	000	1	10	0	×	×	01	1	0	0	00
9	000	1	10	0	×	×	10	1	0	0	00
10	000	1	10	0	×	×	01	1	0	0	00
11	000	1	10	1	×	×	01	1	0	0	00

字段直接编码方式采用相容微操作分在不同字段，因此，能最大限度地并行执行微操作，具有较高的并行控制能力，速度较快，此外，由于对同字段的微操作进行了编码，缩短了微命令字段的长度，因而微指令字较短，节省了控存容量。但是，对于多个微命令组合的字段，由于进行了编码，所以执行时需要相应的译码线路，这样，与直接控制法相比，增加了译码线路，加大了一些成本，并多出了一部分译码时间，不过分段后各字段位数少，所以译码对微指令的执行速度影响不大。

3. 字段间接编码法

在字段直接编码法基础上，可通过字段间接编码的方式进一步压缩微指令长度。字段间接编码是指某一字段可以表示多个微命令组，到底代表哪组微命令，则由另一个专门的字段确定。

虽然这种方式可进一步缩短微指令字的长度，节省控存容量，但意义不大，而且译码线路复杂，时间开销大。通常极少使用这种方式。

4. 最少(最短、垂直)编码法

其基本思想是：采用类似指令编码的思想，即整个微操作码部分作为一个字段，每次只产生一个微操作。采用这种方式得到的微操作码位数最少，长度最短，所以有时称为最少(最短)编码法，用这种格式的微指令编写的微程序较长，所以也称为垂直型微指令。

5. 微指令格式

微指令格式分为两种：水平型微指令和垂直型微指令。通常把采用直接控制方式、字段直(间)接编码方式编码的微指令称为水平型微指令。

显然水平型微指令能最大限度地表示微操作的并行性，但需要使用较长的代码，少则几

十位，多则上百位。较长代码能充分利用硬件并行性所带来速度上的潜在优势，也使微程序中所包含的微指令条数减至最少，所以适用于要求较高速度的场合。但是，一般说来，水平型微指令的编码空间利用率较低，并且编制最佳水平微程序难度较大。

垂直型微指令采用短格式，一条微指令只能控制一、二个微操作，其格式与普通机器指令相仿，设有微操作码字段，由微操作码确定微指令的功能。它所包含的地址码用来指定微操作数所在的寄存器地址或微指令转移地址，也可表示立即数或标志码等。采用垂直型微指令编写的微程序称垂直微程序。垂直微程序不着重于微操作的并行性，并以此换取较短的微指令长度和较高的编码空间利用率，编写垂直微程序的方法和传统的程序设计方法更为接近。因此，垂直型微指令面向算法描述而水平型微指令面向处理机内部控制逻辑的描述。

6.4.4 微指令地址的确定

当前微指令执行结束后，必须确定下一条执行的微指令。下条微指令可能是以下三种情况之一。

(1) 取指微程序的首条微指令。每条机器指令执行完，需要转到取指令微程序执行。所以，若当前执行的是某条机器指令对应的最后一条微指令，则下条微指令就是取指微程序的第一条微指令。

(2) 某机器指令对应微程序的首条微指令。当执行完取指微程序的最后一条微指令，则需要根据当前指令的译码结果，确定下面该执行哪条指令对应的微程序。因此，首先要找到对应微程序的首条微指令。

(3) 某个微程序执行过程中的一条微指令。这又有三种不同的情况。

① 按顺序取出下条微指令执行。

② 无条件转到另一处微指令执行。

③ 根据条件码或指令操作码 op(包括功能码 func)选择不同分支处的微指令执行。

对于上述各种情况，如何来解决微指令执行的顺序控制问题呢？总体来说，是通过在微指令中明显或隐含地指定下条微指令在控存中的地址(简称下条微地址)来解决的。

下条微指令地址的确定方式有两种：计数器法(增量法)和断定法(下址字段法)。

1. 计数器法

计数器法的主要思想是：使用一个专门的微程序计数器 μPC，将下条微指令地址隐含地存放在 μPC 中，因此，这种方法称为计数器法。顺序执行时，根据 μPC+1→μPC，得到下条微指令地址；转移执行时，在当前微指令后添加一条“转移微指令”，并在微指令中添加专门的“转移控制字段”，将“转移微指令”或“转移控制字段”中的控制信息送到微指令地址发生器，与相应的指令操作码以及条件码等组合，生成转移地址送 μPC。前面图 6.38 所示的就是采用计数器法的微程序控制器基本结构。当转移分支很多时，相应的微地址生成逻辑电路很复杂。为简化微地址生成逻辑，通常采用 PLA 或 ROM 来实现。

2. 断定法

计数器法的缺点是必须在不连续执行的微指令之间加入“转移微指令”，这样，在增加微指令条数的同时，还严重影响指令执行速度。如果在微指令中直接明确指定下一条微指令地址，这样，相当于每条都是转移微指令，即使不连续执行也没有关系。这种方法称为断定

法，也称为下址字段法。

断定法虽然加快了指令执行速度，但因为增加了微指令的长度，从而影响控制存储器的有效利用。例如，假定一个采用微程序控制器的处理器的控存容量为4KB，共有500条左右的微指令，这意味着下地址字段至少有9位，微指令长度为64位左右，其中除了下地址字段以外，还有一些其他如控制多分支转移的条件测试和转移控制字段等，也都用于控制微指令的寻址，因此，大约有五分之一的控存空间用于微指令寻址，真正用来存放微命令的空间只有五分之四左右。

此方法最明显的优点是消除了专门的转移微指令，而且在给微指令分配地址时不需要考虑如何排列，也不需要对μPC增量，而用一个简单的微指令地址寄存器（μAR）来存放当前微地址。采用断定法的微程序控制器结构如图6.39所示，其中，微地址修改逻辑根据当前指令、状态条件、下地址字段和转移控制字段来确定微程序的执行顺序。

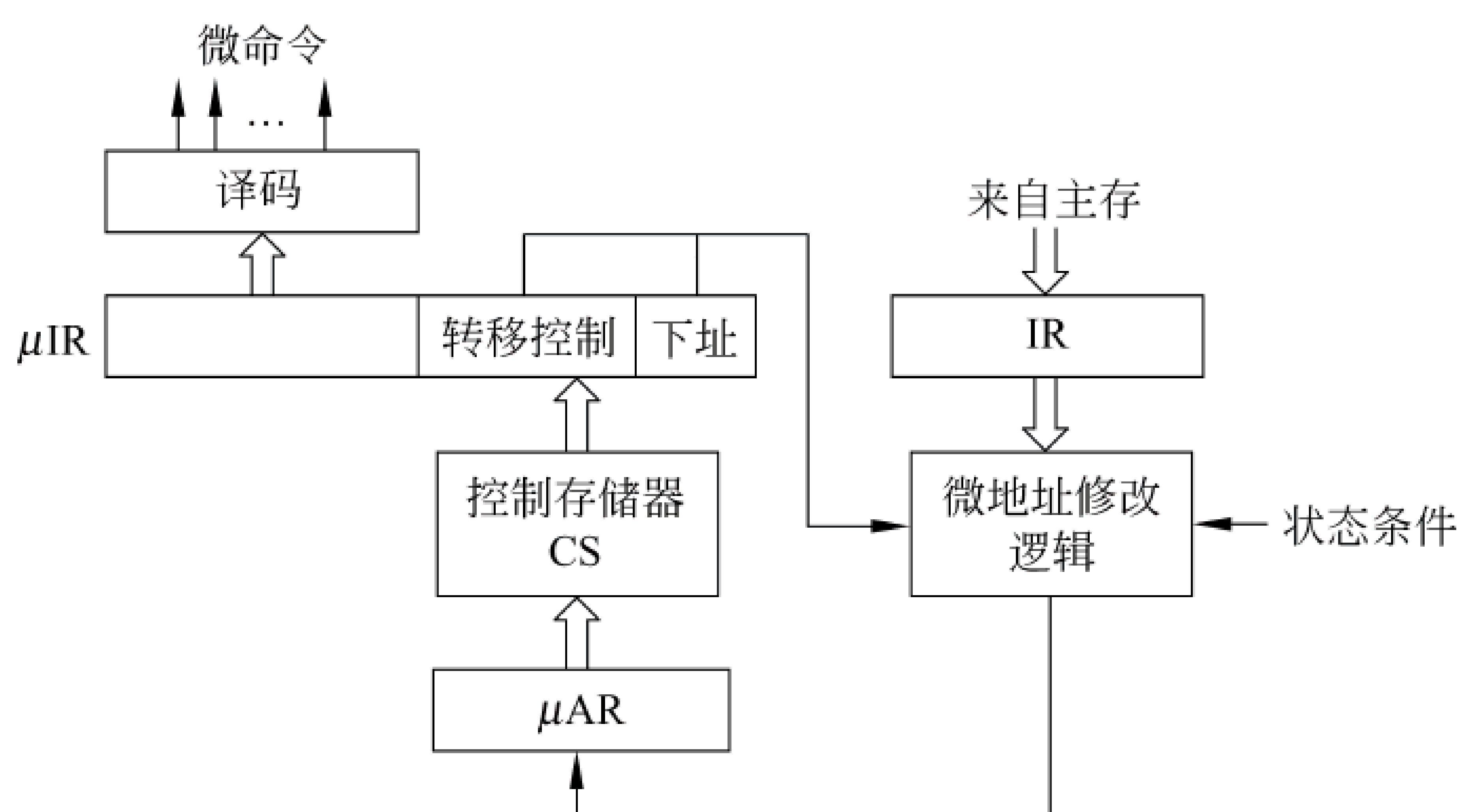


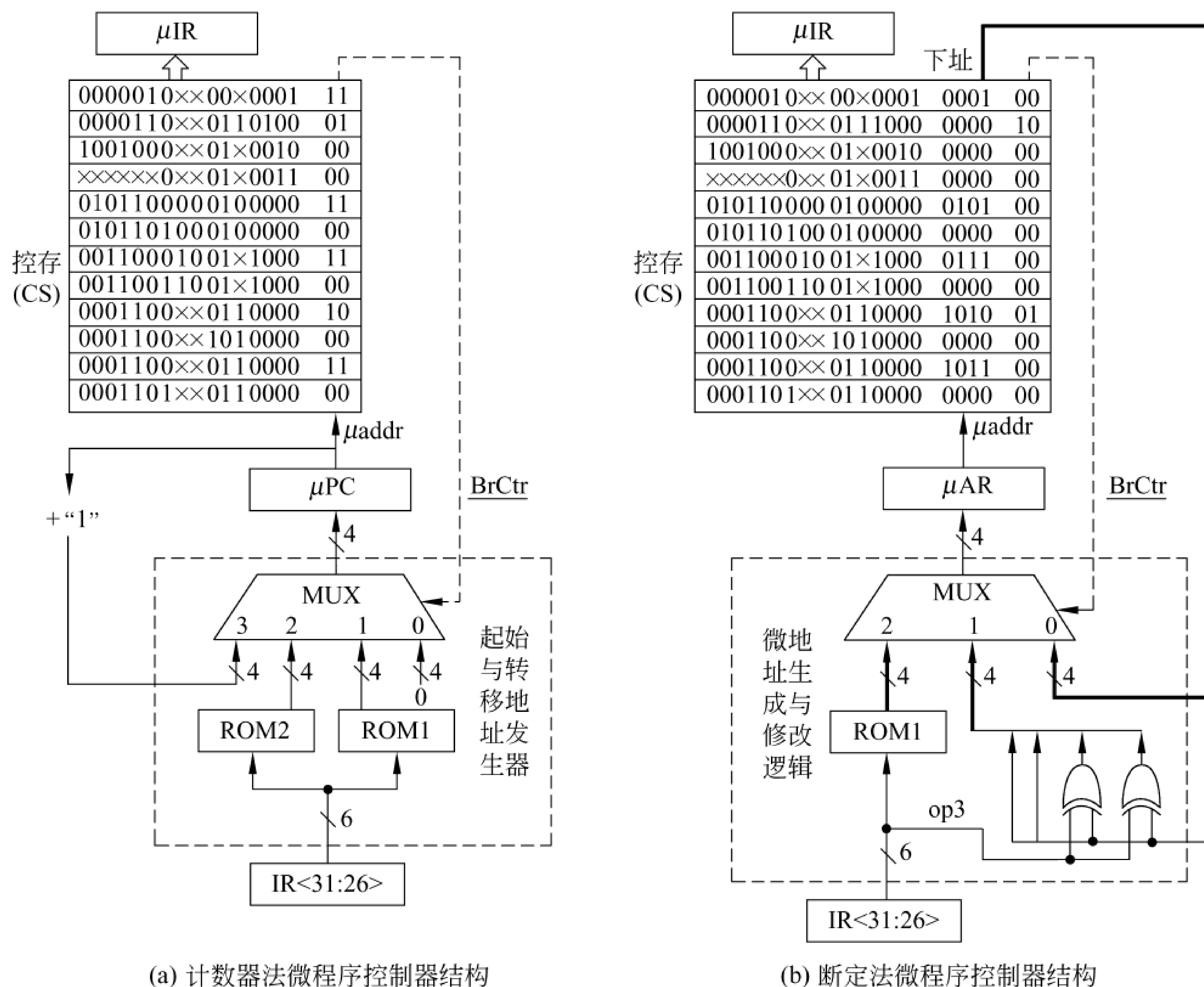
图6.39　采用断定法的微程序控制器结构

下面通过具体例子来介绍计数器法和断定法的相关实现细节。

例6.4　假定采用在微指令字中增加“转移控制字段”的方式来实现分支，并用ROM方式实现机器指令微程序首地址的生成，那么，对于表6.11给出的微程序，请分别给出采用计数器法和下址字段法时的微程序控制器结构。

解：表6.11的微程序中，将状态号作为微指令地址，所以微地址共4位。其中，取指令微程序首址为0000，地址0001处的微指令执行结束后，可能会转到不同指令对应的微程序执行，其分支转移功能由ROM1实现，可能转到beq、jump、ori、R-型、lw/sw这5种指令对应的微程序去执行，beq和jump指令的微程序都只有一条微指令，分别存放在地址0010和0011中；ori和R-型指令的微程序都有两条微指令，其首地址分别为0100和0110；lw/sw指令的微程序有4条微指令，首地址为1000，微程序中有一个分支转移点，需要根据当前指令是lw还是sw来决定1000处的微指令执行后是转到1010处执行还是1001处执行。

图6.40(a)给出的是计数器法微程序控制器的结构。图中μPC具有自增功能，每来一个时钟自动加1。lw/sw指令微程序中的分支功能由ROM2实现。“BrCtr”微命令是在微指令字中增加的转移控制字段，用来控制下条微地址的选择方式。BrCtr的含义如下。



(a) 计数器法微程序控制器结构　　(b) 断定法微程序控制器结构

图 6.40　表 6.11 中的微程序的两种实现方式

BrCtr=00：转到取指微程序首条微指令执行。

BrCtr=01：转到由 ROM1 的输出所指的微指令执行。

BrCtr=10：转到由 ROM2 的输出所指的微指令执行。

BrCtr=11：按顺序执行。

图 6.40(b)给出的是断定法微程序控制器的结构。下条微指令地址可直接来自当前微指令字的下址字段,但在某些情况下需要对其进行修改。例如,lw/sw 指令微程序中的分支功能由操作码 IR<31：26>(对应 op5～op0)中的 op3 对微地址修改实现。因为 OP(lw)=100011,OP(sw)=101011,两者的差别仅在 op3。假定 1000 处微指令的下址字段为 1010,则当 op3=0 时不需修改微地址;当 op3=1 时需将 1010 修改为 1001,所以,只要将 4 位地址的后两位与 op3 进行“异或”即可。“BrCtr”微命令是在微指令字中增加的转移控制字段,用来控制下条微地址的选择方式。BrCtr 的含义如下。

BrCtr=00：转到微指令的下址字段指出的微指令执行。

BrCtr=01：转到由 op3 修改后的微地址所指微指令执行。

BrCtr=10：转到由 ROM1 的输出所指的微指令执行。

图 6.40 的计数器法和断定法中都用 ROM 方式实现多分支转移,ROM1 中存放的是各指令微程序的首地址;ROM2 中只有两个单元,分别存放 1001 和 1010。断定方式下微指令长度为 22 位,计数器法的微指令长度为 18 位。

6.5 异常和中断处理

6.5.1 基本概念

在程序正常执行过程中,CPU 会遇到一些特殊情况而无法继续执行当前程序。这种打断程序正常执行的情况主要有两大类。

1. 内部异常(Exception)

内部异常是指由处理器内部异常引起的意外事件。根据其发生的原因又分为硬故障中断和程序性异常。硬故障中断是由硬连线路出现异常引起的,如电源掉电、存储器线路错等;程序性异常也称软中断,由 CPU 执行某个指令而引起的发生在处理器内部的异常事件,也称为例外。如除数为 0、溢出、断点、单步跟踪、寻址错、访问超时、非法操作码、堆栈溢出、缺页、地址越界、数据格式错等。按发生异常的报告方式和返回方式的不同,内部异常可分为故障(Fault)、自陷(Trap)和终止(Abort)三类。

(1) 故障(Fault)

故障也称为失效,它是在引起故障的指令启动后、执行结束前被检测到的一类异常事件。例如,指令译码时,出现“非法操作码”;取指令或数据时,发生“段不存在”、“缺页”或“保护错”;执行除法指令时,发现“除数为 0”等。显然,“段不存在”、“缺页”等这类异常处理后,已将需要的段或页面从磁盘调到主存,所以可继续回到发生故障的指令继续执行,此时,断点为当前发生故障的指令;对于“非法操作码”、“保护错”等,因为无法通过异常处理程序恢复故障,因此不能回到原断点继续执行,必须终止进程的执行;对于“除数为 0”,可有不同处理方式,可将指令执行结果用特殊的值(如∞或 NaN)来表示后继续回原断点执行,也可终止进程执行。

(2) 自陷(Trap)

自陷也称为陷阱或陷入,与“故障”等其他意外发生的异常事件不同,是预先安排的一种“异常”事件,就像预先设定的“陷阱”一样,首先通过某种方式将 CPU 设定为处于某个特定状态,在程序执行过程中,一旦某条指令的执行发生了相应状态所满足的条件,则 CPU 调出特定的程序进行相应的处理。

通常的做法是事先在程序中用一条特殊指令或通过某种方式设定特殊控制标志来人为设置一个“陷阱”,当执行到被设置了“陷阱”的指令时,CPU 在执行完“陷阱指令”后,自动根据不同“陷阱”类型进行相应的处理,然后返回到“陷阱指令”的下一条指令执行。注意,当“陷阱指令”是转移指令时,并不能返回到下一条指令执行,而是返回到转移目标指令执行。

例如,在 Intel 80x86 体系结构中,可以执行指令 STI 使 CPU 处在开中断状态(即中断标志 IF=1),并通过 PUSHF/PUSHFD 和 POPF/POPFD 指令将 CPU 设置为跟踪状态(即跟踪标志 TF=1)。这样,CPU 便处于单步跟踪状态(TF=1 且 IF=1),此时,每条指令都被设置成了“陷阱”,CPU 在执行每条指令后,都会转去执行一个特定的“单步跟踪处理程序”,该程序将当前指令执行的结果显示在屏幕上。单步跟踪处理过程中,CPU 会自动把标

志寄存器压栈，然后清除 TF 和 IF。这样，在单步跟踪处理时，CPU 就能以正常方式工作。单步处理结束、返回到下条指令执行之前，再从栈中取出标志，恢复 TF 和 IF 的值，使 CPU 回到单步跟踪状态，这样，下条指令又是“陷阱”指令，将被跟踪执行。如此这样下去，每条指令都被跟踪执行，直到将 TF 和 IF 清 0 为止。在 80x86 中，用于程序调试的“断点设置”功能就是通过“自陷”方式来实现的。

此外，还有系统调用指令、条件陷阱指令（如 MIPS 中的 teq、teqi、tne、tnei 等一组按条件进入陷阱的指令）等都属于自陷指令，执行到这些指令时，无条件或有条件地自动调出操作系统内核程序进行执行。

（3）终止（Abort）

如果在执行指令过程中发生了使机器无法继续执行的硬件故障。如电源掉电，线路故障等，则程序将无法继续执行，只好终止，此时，调出中断服务程序来重启系统。这种异常与故障和自陷不同，不是由特定指令产生的，而是随机发生的，所以无法确定发生异常的是哪条指令。

2. 外部中断（Interrupt）

程序执行过程中，若外设完成任务或发生某些特殊事件（如打印机缺纸、定时采样计数时间到、键盘缓冲满等），会向 CPU 发中断请求，要求 CPU 对这些情况进行处理。通常，每条指令执行完后，CPU 都会主动去查询有没有“中断”请求，有的话，则将下条指令地址作为断点保存，然后转到相应的“中断服务程序”执行，结束后回到断点继续执行。

这种事件与执行的指令无关，和指令的执行异步发生，是由 CPU 外部的 I/O 发出的，所以，称为 I/O 中断或外部中断，需要通过外部中断请求线向 CPU 请求。

不同的计算机体系结构和教科书对“异常”（Exception）和“中断”（Interrupt）定义的内涵不同，例如 Power PC 体系结构用“异常”表示各种意外事件，而用“中断”表示控制流被改变；8086/8088 微处理器不区分“异常”和“中断”，把两者统称为“中断”，由 CPU 内部产生的异常称为“内中断”；通过中断请求线 INTR 和 NMI 从 CPU 外部发出的中断请求为“外中断”。内中断皆为不可屏蔽中断，通过 INTR 信号线发出的外中断是可屏蔽中断，而通过 NMI 信号发出的是不可屏蔽中断，它们主要是一些重要或紧急的硬件故障，如电源掉电、存储器线路错等。不可屏蔽中断的处理优先级最高，任何时候只要发生不可屏蔽中断，都要中止现行程序的执行，转到不可屏蔽中断处理程序执行。每个中断都有一个类型号，每个中断类型号都对应一个中断服务程序，其入口地址放在一个专门的中断向量表中。例如，类型 0 为“除法错”，类型 2 为“NMI 中断”，类型 15 为“缺页”等。前 32 个中断类型（00H～1FH）保留给处理器使用，剩余的可以由用户自行定义功能，通过执行“INT n”指令（第二字节给出中断类型号：n=32～255），使 CPU 自动转到用户编写的中断服务程序执行，“INT n”也称为软中断指令。

从 80286 开始，Intel 统一把“内中断”称为内部异常，而把“外中断”称为外部中断。为了区分内部异常和外部中断，本教材将 CPU 内部异常事件称为“异常（Exception）”，而将外部中断事件称为“中断（Interrupt）”。中断是计算机外设的一种输入输出方式，所以有关中断的主要内容放到后面第 9 章介绍。本章后面主要介绍异常处理。实际上，两者的处理过程和基本处理方式是类似的。

6.5.2 异常处理过程

在CPU执行指令过程中，如果发生了异常事件，则CPU必须进行相应的处理。CPU对异常的处理过程可分为以下两个步骤：保护断点和程序状态、识别异常事件并转异常处理。

1. 保护断点和程序状态

前面提到，对于不同的异常事件，其返回地址（即断点）不同。例如，“故障”异常的断点是发生故障的当前指令的地址；“自陷”异常的断点则是自陷指令后面一条指令的地址。显然，断点的值由异常类型和发生异常时PC的值决定。例如，对于图6.33的多周期数据通路，如果在执行lw/sw指令时发生“缺页”异常，则其断点值应该是当前PC值减4。因为，在发现“缺页”时，已在取指令状态执行了PC+4，所以，PC必须减4才能保证“缺页”异常处理返回后重新执行lw/sw指令。

为了能在异常处理后正确返回到原来被中断的程序继续执行，数据通路必须能正确计算断点值。假定计算出的断点值存放在PC中，则保护断点时，只要将PC送到堆栈或特定的寄存器中即可。

为了能够支持异常或中断的嵌套处理，大多数处理器将断点保存在堆栈中，如80x86处理器的断点被保存在堆栈。如果系统不支持多重中断嵌套处理，则可以将断点保存在特定寄存器中，而不需送到堆栈保存，如MIPS处理器用EPC寄存器专门存放断点。

因为异常处理后可能还要回到原被中断的程序继续执行，所以，被中断时原程序的状态（如产生的各种标志信息、允许自陷标志等）都必须保存起来。通常每个正在运行程序的状态信息存放在一个专门的寄存器中，这个专门寄存器统称为程序状态字寄存器（PSWR），存放在PSWR中的信息称为程序状态字（PSW-Program Status Word）。例如，在80x86中，程序状态字寄存器就是标志寄存器EFLAGS。与返回地址一样，PSW也要被保存到堆栈或特定寄存器中，在异常返回时，将保存的PSW恢复到PSWR中。当然，如果处理器体系结构中规定PSWR是指令可直接访问的寄存器，则PSW的保存/恢复也可以和现场的保存/恢复一样，由异常处理程序（或中断服务程序）来实现，而无需硬件自动保存。

2. 识别异常事件并转异常处理

在调出异常处理程序之前，必须知道发生了什么异常。一般来说，内部异常事件和外部中断源的识别方式不同，大多数处理器会将两者分开来处理。内部异常事件的识别大多采用软件识别方式，而外部中断源则可以采用软件识别或硬件识别方式。

软件识别方式是指，CPU设置一个异常状态寄存器，用于记录异常原因。操作系统使用一个统一的异常查询程序，该程序按一定的优先级顺序查询异常状态寄存器，先查询到的异常先被处理。例如，MIPS就采用软件识别方式，CPU中有一个cause寄存器，位于地址0x8000 0180处有一个专门的异常查询程序，它通过查询cause寄存器来检测异常类型，然后转到内核中相应的异常处理程序进行具体的处理。

因为像“故障”和“陷阱”之类的内部异常通常是在执行某条指令时发现的，可以通过对指令执行过程中某些条件的判断来发现是否发生了“异常”，而且一旦发现可以马上进行处理，所以，内部异常事件也可以不通过专门的查询程序来识别，而在发现“异常”时直接得到异常错误代码，根据不同的错误代码，转到相应的异常处理程序即可。80x86的处理方式就

是这样。

由于外部中断的发生与CPU正在执行的指令没有必然联系，相对于指令来说，外部中断是随机的、与指令无关的。所以，并不能根据指令执行过程中的某些现象来判断是否发生了"中断"请求。因此，对于外部中断，只能在每条指令执行完后、取下条指令之前去查询是否有中断请求。通常CPU通过采样对应的中断请求引脚线来进行查询，如果发现"中断请求"线有效，则说明有中断请求，但是到底是哪个中断源发出的请求，还需要进一步识别。有关外部中断源的响应、识别和中断处理程序的结构等内容，在第9章中将详细介绍。

假定在执行异常处理程序时，又发生了新的异常，怎么办？在发现异常并转到异常处理程序的过程中，若在保存正在运行进程的状态时又发生新的异常，那么，就会因为要处理新的异常，进而把原来进程的状态和保存的返回断点破坏掉，因此，应该有一种机制来"禁止"在响应并处理异常时再响应新的异常。通常通过设置"中断/异常允许"状态位(或"中断/异常允许"触发器)来实现。当"中断/异常允许"状态位置1，则"开中断/异常"，表示允许响应中断/异常；若"中断/异常允许"状态位被清0，则"关中断/异常 "表示不允许响应中断/异常。一般由OS通过管态指令来设置该位的状态。例如，在80x86体系结构中，可以执行指令STI或CLI，使标志寄存器EFLAGS中的IF位被设置为1或清0，使CPU处在开或关中断状态。80x86规定，异常处理过程中，IF必须清0，以禁止响应新的异常或中断。

*6.5.3 带异常处理的处理器设计

异常和中断处理是处理器设计中最具挑战性的任务之一，为了说明CPU如何处理异常和中断，本教材以MIPS为例简单说明带异常处理的数据通路的设计。

为简单起见，MIPS中的断点信息保存到一个特殊的32位寄存器EPC中。写入EPC的断点值可能是正在执行中的异常指令的地址(故障时)，也可能是异常指令的下条指令的地址(自陷)。前者需要把PC的值减4后送到EPC，后者则直接送PC到EPC。

MIPS采用软件识别方式，用一个专门的32位寄存器(cause)来记录异常原因，异常查询程序的入口地址为0x8000 0180，该异常查询程序将根据cause的值判断发生了何种异常，然后根据异常类型控制转到相应的异常处理程序执行。

假定处理器能处理的异常类型有两种：非法操作码(cause=0)和溢出(cause=1)，则在图6.33的多周期数据通路中加入相应的异常处理后，得到带异常处理的多周期数据通路如图6.41所示。其中右部加黑加粗部件是与异常处理相关的部分。

图6.41中对两个寄存器EPC和cause分别加入了以下两个"写使能"控制信号。

EPCWr：在需要保存断点时，该信号有效，使断点值PC写入EPC。

CauseWr：在CPU发现异常(如非法指令、溢出)时，该信号有效，使异常类型被写到cause寄存器。

此外，还需要一个控制信号IntCause来选择将正确的值写入到cause中。

对于异常程序的切换，需要将异常查询程序的入口地址(MIPS为0x8000 0180)写入PC，因此，图中在原来PCSource控制的多路选择器中又增加了一路，其输入为0x8000 0180，用PCSource=11来控制选择。

为了支持图6.41所示的数据通路，必须对图6.33所示数据通路对应的控制器进行修改。可以在原图6.34所示的有限状态机中增加异常处理状态，每种异常占一个状态。

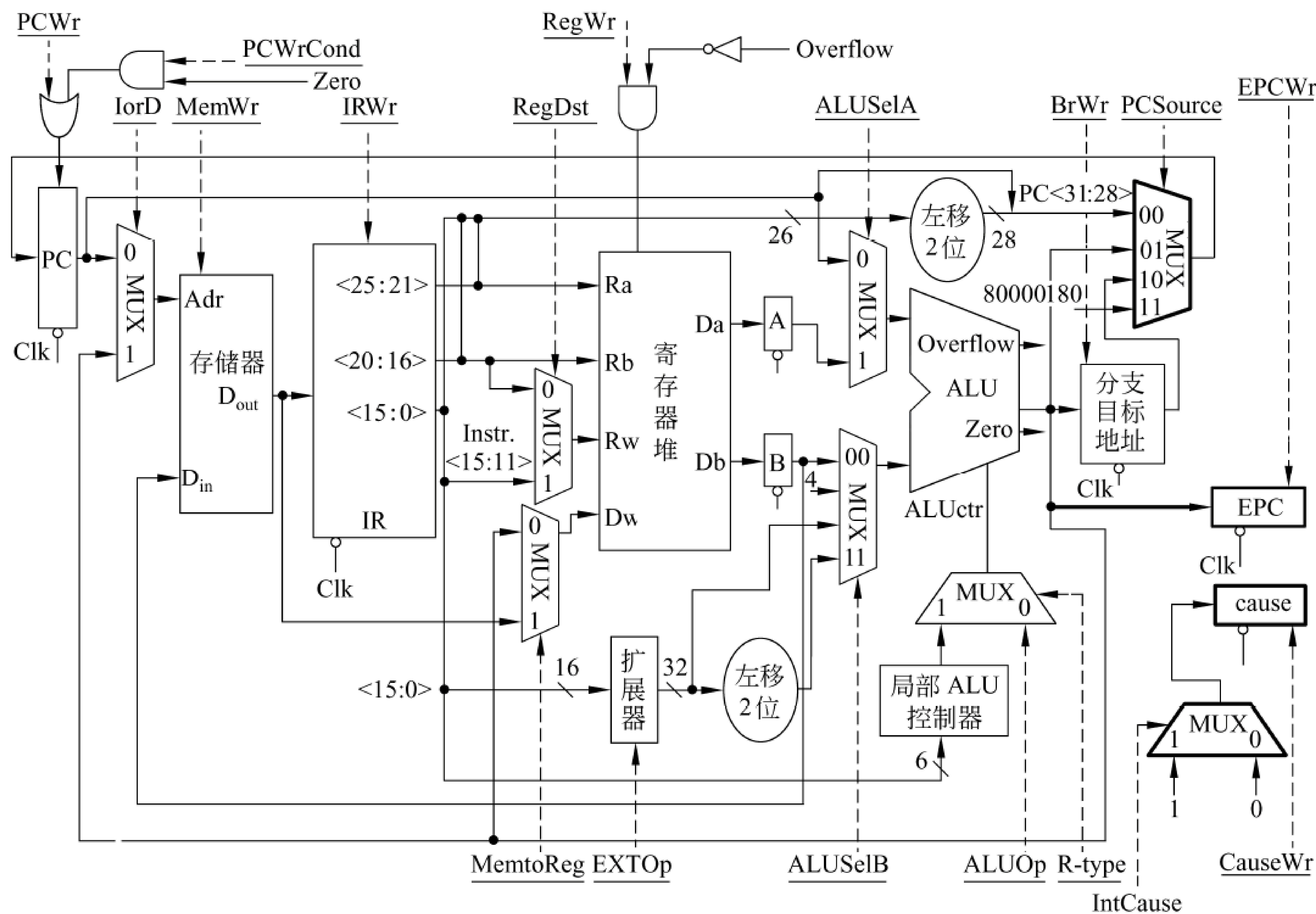


图 6.41　带异常处理的多周期数据通路

带异常处理的有限状态机如图 6.42 所示。

图 6.34 所示的有限状态机中已有状态 0～状态 11，因此，将两种异常处理对应的状态分别用状态 12 和状态 13 来表示(加黑加粗的两个状态)。

(1) 状态 12。“非法操作码”异常。若在状态 1 进行指令译码时发现 op 字段是一个未定义的编码，则进入状态 12，其控制信号用来控制完成以下操作：①将 0 送 cause 寄存器；②PC 减 4 送 EPC；③将 0x8000 0180 送 PC。

(2) 状态 13。“溢出”异常。当 R-型指令或 I-型运算类指令执行后在 ALU 输出端的 Overflow 为 1 时，则进入状态 13。其控制信号用来控制完成以下操作：①将 1 送 cause 寄存器；②PC 减 4 送 EPC；③将 0x8000 0180 送 PC。

根据图 6.42 所示的有限状态机不难实现相应的控制逻辑。

同上述两种异常的处理一样，对于“缺页(Page Fault)”异常的处理，只要在每次存储器访问过程中，查询其对应页表项中的“有效位(Valid)”是否为 1。若不为 1，则转入“缺页”异常状态，该状态中的控制信号控制数据通路完成以下操作：①将相应的值送 cause 寄存器；②PC 减4 送 EPC(“缺页”处理结束后回到发生异常的指令重新执行一次)；③将 0x8000 0180 送 PC。

“缺页”异常一定要在发生缺失的存储器操作时钟周期内捕获到，并在下个时钟转到异常处理，否则，会发生错误。例如：对于指令 lw \$1, 0(\$1)，若没有及时捕获“缺页”异常并转到异常处理，而使 \$1 发生了改变，那么，再重新回到该指令执行时，所读的内存单元地

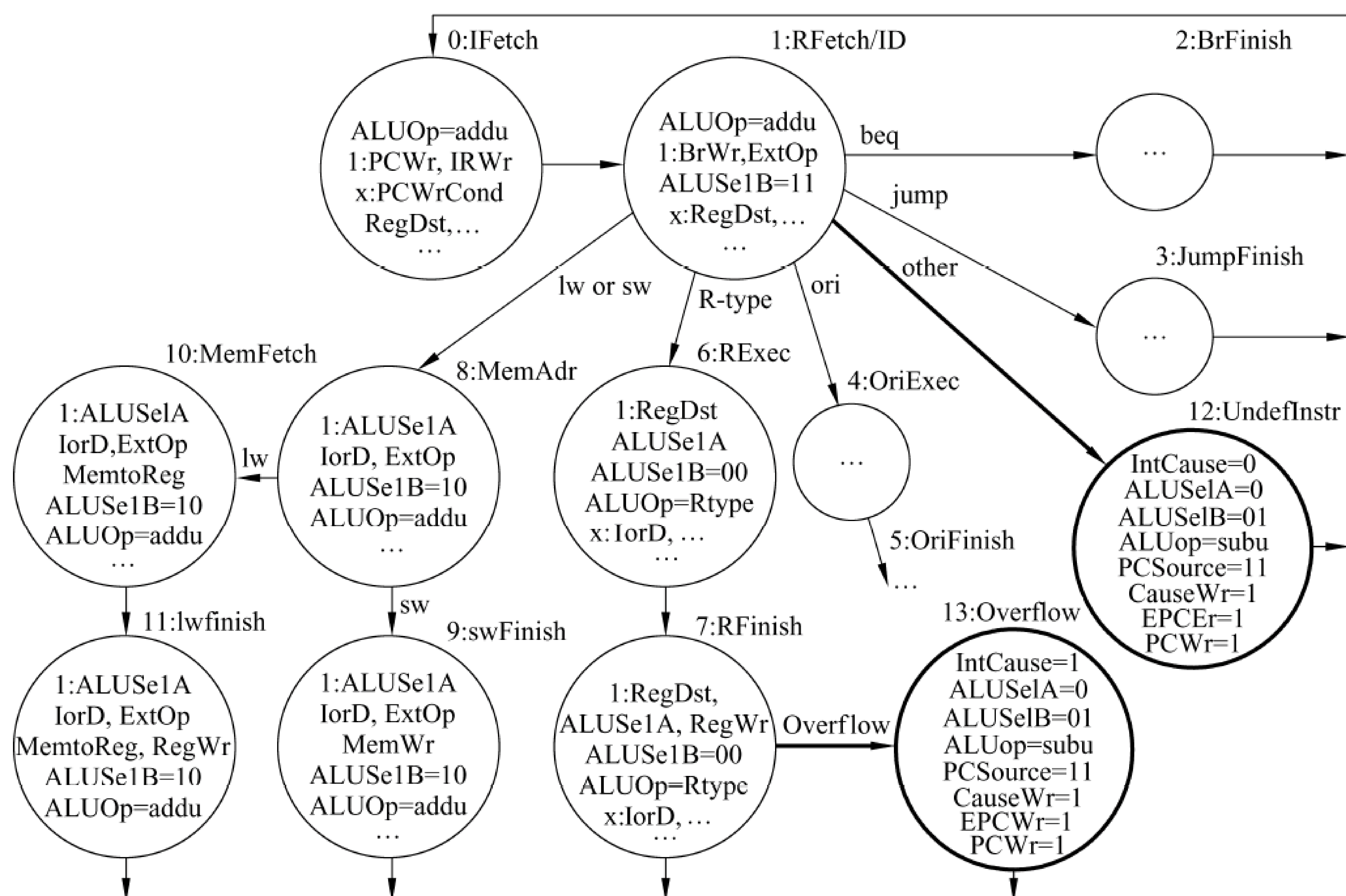


图 6.42　带异常处理的有限状态机示意图

址被改变，因而会发生严重错误。

在“缺页”异常处理程序中，应该完成以下功能。

(1) 保存正在运行进程的所有状态，包括程序可见寄存器、页表地址寄存器、EPC 和 cause 寄存器等。

(2) 确定缺失的是指令还是数据。若是指令，则发生在取指令阶段(EPC 中的地址是缺失地址)；若是数据，则发生在取数或存数阶段。

(3) 完成磁盘读写操作。查找页表项，得到磁盘地址，找到盘上被访问页的位置，选择被替换的页，若被替换页已被修改，则在把新页替换到内存前，先要把内存中淘汰的页写回磁盘。启动磁盘读操作，把被访问页调到所选择的物理页框中。因为磁盘读(替换时还要写磁盘)需要花费几百万个时钟周期，所以，此时，OS 会挂起当前进程，把另一个进程调到处理器执行，直到磁盘访问结束。

(4) 恢复引起缺页的进程的状态。即将第(1)步保存的进程状态取到相应的寄存器中。

(5) 执行“从异常返回”指令。修改机器状态使 CPU 从核心态转到用户态，并将 EPC 中的地址装到 PC 中。

对于不同计算机体系结构以及不同的指令，其涉及到的访存次数可能不同。简单指令最多只要访问一次内存去取数据，但有些复杂指令系统中的复杂指令，可能会多次访问内存去取数据。因而，缺页异常可能发生在一条指令执行过程中的不同时刻，所以异常处理返回后应回到不同的地方继续执行。对于访问指令时发生的缺页，可以回到发生缺页的指令重新执行。而对于访问数据时发生的缺页，简单指令(仅一次访存)的情况下，可强迫指令结束，重新执行发生缺页的指令；复杂指令(多次访存)情况下，由于可能发生多次缺页，指令

被中止在中间某个阶段，缺页处理后，可从中间阶段继续执行，因而，需要能够保存和恢复指令执行过程中的机器状态。

6.6 本章小结

本章以 MIPS 指令系统中具有代表性的 11 条指令作为实现目标，介绍了单周期数据通路和多周期数据通路的设计过程以及相应的控制部件的设计。主要内容包括 CPU 的主要功能和内部结构、指令的执行过程、数据通路的基本组成、数据通路的定时、数据通路中信息的流动过程、控制器的实现方式、硬连线路控制器的设计、微程序控制器的设计、异常和中断的概念等。具体总结如下。

- CPU 的基本功能：周而复始地执行指令，并处理异常和中断，具体功能如下。
  - 控制程序的执行顺序。
  - 控制指令进行什么操作。
  - 控制每个操作何时进行。
  - 对数据进行算术或逻辑运算。
  - 控制对存储器或 I/O 的访问。
  - 判断有无异常或中断并调出相应处理程序。
- CPU 的基本结构：由数据通路(Datapath)和控制单元(Control unit)组成。
  - 数据通路中包含组合逻辑单元和存储信息的状态单元。组合逻辑单元用于对数据进行处理，如加法器、运算器 ALU、扩展器(0 扩展或符号扩展)、多路选择器以及状态单元的读操作逻辑等。状态单元包括触发器、寄存器等，用于对指令执行的中间状态或最终结果进行保存。
  - 控制单元对取出的指令进行译码，与指令执行得到的条件码或当前机器的状态、时序信号等组合，生成对数据通路进行控制的控制信号。
- CPU 中的寄存器
  - 用户可见寄存器：用户程序中的指令可直接访问或间接修改其值的寄存器。
    - 通用寄存器：可用来存放地址或数据。
    - 地址寄存器：专门用来存放首地址或指针信息。如：段寄存器、变址寄存器、基址寄存器、堆栈指针、帧指针等。
    - 程序计数器 PC：存放当前或下条指令的地址。
  - 用户部分可见寄存器：用户程序中的指令只能读取部分信息的寄存器，如程序状态字寄存器 PSWR 或标志(条件码)寄存器，其内容由 CPU 根据指令执行结果自动设定，用户程序的指令只能隐含读出部分内容，而不能修改寄存器的内容。
  - 用户不可见寄存器：用于记录控制和状态信息的寄存器，只能由 CPU 硬件或操作系统的内核程序访问，用户程序不可访问。
    - 指令寄存器 IR：存放正在执行的指令。
    - 存储器地址寄存器 MAR：存放将要访问的存储单元的地址。
    - 存储器缓冲(数据)寄存器 MBR(MDR)：存放从存储器读出或写入存储器的数据。

- ▲ 其他寄存器：如中断请求寄存器、进程控制块指针、系统堆栈指针、页表基址寄存器等。

- 指令执行过程：取指、译码、取数、运算、存结果、查中断。
  - 指令周期：取出并执行一条指令的时间，由若干个机器周期或直接由若干个时钟周期组成。
  - 机器周期：通过一次总线事务来访问一次主存或 I/O 的时间，一个机器周期由多个时钟周期组成（但是现代计算机已经没有机器周期的概念，每个指令周期直接由若干个时钟周期组成）。
  - 时钟周期：时钟是 CPU 中用于控制信号同步的信号，时钟周期是 CPU 中最小的时间单位。
- 数据通路的定时方式。现代计算机都采用时钟信号进行定时，一旦时钟边沿信号到来，数据通路中的状态单元开始写入信息。
- 数据通路中信息的流动过程。每条指令的功能不同，所以，每条指令执行时数据在数据通路中所经过的部件和路径可能不同。但是，每条指令在取指令阶段都一样。
- 控制单元的实现方式。根据不同的控制描述方式，可以有两种不同的实现方式。
  - 硬连线路控制器：将指令执行过程中每个时钟周期所包含的控制信号取值组合看成一个状态，每来一个时钟，控制信号会有一组新的取值，也就是一个新的状态，这样，所有指令的执行过程就可以用一个有限状态转换图来描述。实现时，用一个组合逻辑电路（一般为 PLA 电路）来生成控制信号，用一个状态寄存器实现状态之间的转换。
  - 微程序控制器：将指令执行过程中每个时钟周期所包含的控制信号取值组合看成是一个 0/1 序列，每个控制信号对应一个微命令，控制信号取不同的值，就发出不同的微命令。这样，若干微命令组合成一个微指令，每条指令所包含的动作就由若干条微指令来完成。每条指令执行时，先找到对应的第一条微指令，然后按照特定的顺序取出后续的微指令执行。每来一个时钟，执行一条微指令。实现时，每条指令对应的微指令序列事先存放在一个只读存储器（称为控制存储器，简称控存）中，用一个 PLA 电路或 ROM 来生成每条指令对应的微程序的第一条微指令地址，用相应的微程序定序器来控制微指令执行流程。微程序定序器的实现有计数器法和断定法（即下址字段法）两种。
- 异常和中断
  - 异常（内中断）：指 CPU 内部在执行某条指令时发生的程序异常或硬件异常，有故障、陷阱和终止三种类型，也被称为程序性中断或软中断。
    - ▲ 故障：由某条正在执行的指令产生的异常，如“溢出”、“除数为 0”、“非法操作码”、“缺页”、“地址越界”、“访问越权”、“段不存在”、“堆栈溢出”等，有些故障修复后程序可以继续执行下去，有些故障不能修复，只能中止发生异常的程序的执行。
    - ▲ 陷阱：是预先安排的一种“异常”事件，例如，断点设置、单步跟踪、系统调用、条件自陷等引起的陷阱。
    - ▲ 终止：严重的硬件故障，一旦发生只能终止整个系统的运行，重启系统。

◆ 外部中断：外设或它机通过中断请求线向 CPU 提出的处理请求，与指令无关，是一种 I/O 方式。

习　题　6

1. 给出以下概念的解释说明。

(1) 指令周期	(2) 机器周期	(3) 同步系统	(4) 时序信号
(5) 控制单元	(6) 执行部件	(7) 操作元件	(8) 状态元件
(9) 多路选择器	(10) 扩展器	(11) 定时方式	(12) 边沿触发
(13) 程序计数器(PC)	(14) 指令寄存器(IR)	(15) 指令译码器(ID)	(16) 时钟周期
(17) 转移目标地址	(18) 控制信号	(19) 硬连线控制器	(20) 微程序控制器
(21) 控制存储器(控存 CS)	(22) 微代码	(23) 微指令	(24) 微程序
(25) 固件	(26) 异常	(27) 中断	(28) 自陷(Trap)
(29) 异常处理程序	(30) 异常/中断允许位	(31) 关中断	(32) 开中断

2. 简单回答下列问题。

(1) CPU 的基本组成和基本功能各是什么？

(2) 取指令部件的功能是什么？

(3) 控制器的功能是什么？

(4) 为什么对存储器按异步方式进行读写时需要 WMFC 信号？按同步方式访问存储器时，CPU 如何实现存储器读写？

(5) 单周期处理器的 CPI 是多少？时钟周期如何确定？为什么单周期处理器的性能差？单周期方式下，在一个指令周期内某个部件能否被重复使用多次？为什么？

(6) 多周期处理器的设计思想是什么？每条指令的 CPI 是否相同？为什么在一个指令周期内某个部件可被重复使用？

(7) 在控制逻辑设计方法上，单周期处理器和多周期处理器的差别是什么？

(8) 硬连线控制器和微程序控制器的特点各是什么？

(9) 为什么 CISC 多用微程序控制器实现，RISC 多用硬连线控制器实现？

(10) 水平型微指令和垂直型微指令的基本概念和优缺点是什么？

(11) CPU 检测内部异常和外部中断的方法有什么不同？

3. 图 6.9 中假定总线传输延迟和 ALU 运算时间分别是 20ps 和 200ps，寄存器建立时间为 10ps，寄存器保持时间很小，可忽略不计，寄存器的锁存延迟(Clk-to-Q Time)为 4ps，控制信号生成的延迟时间(Clk-to-signal Time)为 7ps，三态门接通时间为 3ps，则从当前时钟到达开始算起，完成以下操作的最短时间是多少？各需要几个时钟周期？

(1) 将数据从一个寄存器传送到另一个寄存器

(2) 将程序计数器 PC 加 1

4. 图 6.43 给出了某 CPU 内部结构的一部分，MAR 和 MDR 直接连到存储器总线(图中省略)。在两个总线之间的所有数据传送都需经过算术逻辑部件 ALU。ALU 的部分控制信号及其功能如下。

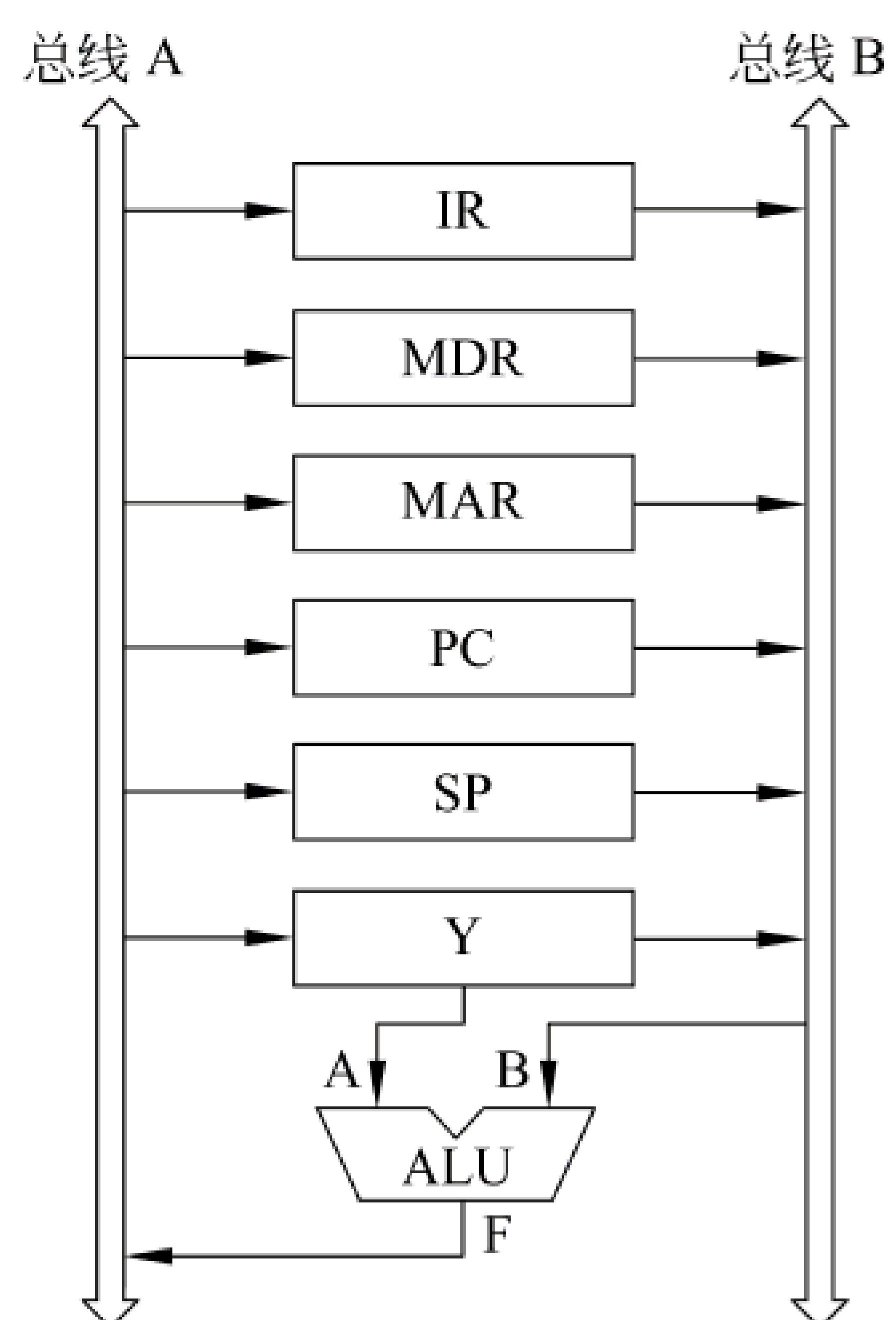


图 6.43　习题 6.4 图示

MOVa: F= A;　MOVb: F=B;

a+1: F=A+1;　b+1: F=B+1

a-1: F=A-1;　b-1: F=B-1

其中 A 和 B 是 ALU 的输入，F 是 ALU 的输出。假定该 CPU 的指令系统中调用指令 CALL 占两个字，第一个字是操作码，第二个字给出子程序的起始地址，返回地址保存在主存的栈中，用 SP(栈指示器)指向栈顶，存储器按字编址，每次按同步方式从主存读取一个字，请写出读取并执行 CALL 指令所要求的控制信号序列(提示：当前指令地址已在PC 中)。

5. 假定某计算机字长 16 位，CPU 内部结构如图 6.9 所示，CPU 和存储器之间采用同步方式通信，按字编址。指令采用定长指令字格式，由两个字组成，第一个字指明操作码和寻址方式，第二个字包含立即数 imm16。若一次存储器访问所用时间为两个 CPU 时钟周期，每次存储器访问存取一个字，取指令阶段第二次访存将 imm16 取到 MDR 中，请写出下列指令在指令执行阶段(不考虑取指令阶段)的控制信号序列，并说明需要几个时钟周期。

(1) 将立即数 imm16 加到寄存器 R1 中，此时，imm16 为立即操作数。

即 R[R1]←R[R1]+imm16。

(2) 将地址为 imm16 的存储单元的内容加到寄存器 R1 中，此时，imm16 为直接地址。

即 R[R1]←R[R1]+M[imm16]。

(3) 将存储单元 imm16 的内容作为地址所指的存储单元的内容加到寄存器 R1 中。此时，imm16 为间接地址。即 R[R1]←R[R1]+M[M[imm16]]。

6. 假定图 6.24 单周期数据通路对应的控制逻辑发生错误，使得控制信号 RegWr、RegDst、Branch、MemWr、ExtOp、R-type 中某一个在任何情况下总是为 0，则该控制信号为 0 时哪些指令不能正确执行？要求分别讨论。

7. 假定图 6.24 单周期数据通路对应的控制逻辑发生错误，使得控制信号 RegWr、RegDst、Branch、MemWr、ExtOp、R-type 中某一个在任何情况下总是为 1，则该控制信号为 1 时哪些指令不能正确执行？要求分别讨论。

8. 要在 MIPS 指令集中增加一条 swap 指令，可以有两种做法。一种做法是采用伪指令方式(即软件方式)，这种情况下，当执行到 swap 指令时，用若干条已有指令构成的指令序列来代替实现；另一种做法是直接改动硬件来实现 swap 指令，这种情况下，当执行到 swap 指令时，则可在 CPU 上直接执行。

(1) 写出用伪指令方式实现“swap rs, rt”时的指令序列(提示：伪指令对应的指令序列中不能使用其他额外寄存器，以免破坏这些寄存器的值)。

(2) 假定用硬件实现 swap 指令时会使每条指令的执行时间增加 10%，则 swap 指令要在程序中占多大的比例才值得用硬件方式来实现？

9. 假定图 6.33 多周期数据通路对应的控制逻辑发生错误，使得控制信号 PCWr、MemtoReg、IRWr、RegWr、BrWr、MemWr、PCWrCond、R-type 中某一个在任何情况下总是为 0，则该控制信号为 0 时哪些指令不能正确执行？要求分别讨论。

10. 假定图 6.33 多周期数据通路对应的控制逻辑发生错误，使得控制信号 PCWr、MemtoReg、IRWr、RegWr、BrWr、MemWr、PCWrCond、R-type 中某一个在任何情况下总是为 1，则该控制信号为 1 时哪些指令不能正确执行？要求分别讨论。

11. 假定有一条 MIPS 伪指令“Bcmp $t1, $t2, $t3”，其功能是实现对两个主存块数据的比较，$t1 和 $t2 中分别存放两个主存块的首地址，$t3 中存放数据块的长度，每个数据占 4 个字节，若所有数据都相等，则将 0 置入 $t1；否则，将第一次出现不相等时的地址分别置入 $t1 和 $t2 并结束比较。若 $t4 和 $t5 是两个空闲寄存器，请给出实现该伪指令的指令序列，并说明在类似于图 6.33 所示的多周期数据通路中执行该伪指令时要用多少时钟周期。

12. 某计算机字长 16 位，标志寄存器 Flag 中的 ZF、NF 和 VF 分别是零标志、符号标志和溢出标志，采

用双字节定长指令字。假定该计算机中有一条 Bgt(大于零转移) 指令,其指令格式为:第一个字节指明操作码和寻址方式,第二个字节为偏移地址 imm8,其功能如下。

若(ZF+(NF⊕VF)=0)则 PC=PC+2+imm8 否则 PC=PC+2。

完成如下要求并回答问题。

(1) 该计算机存储器的编址单位是什么?

(2) 画出实现 Bgt 指令的数据通路。

13. 对于某个 MIPS 多周期处理器,假定将访问数据的过程分成两个时钟周期,则可使时钟频率从 4.8GHz 提高到 5.6GHz,但这样会使得 lw 和 sw 指令增加时钟周期数。已知基准程序 CPUint 2000 中各类指令的频率为:Load:25%,Store:10%,Branch:11%,Jump:2%,ALU:52%。那么,以基准程序 CPUint 2000 为标准,处理器时钟频率提高后的性能提高了多少?若将取指令过程再分成两个时钟周期,则可进一步使时钟频率提高到 6.4GHz,此时,时钟频率的提高是否也能带来处理器性能的提高?为什么?

14. 假设 MIPS 指令系统中有一条 I-型指令"Bgt Rs,Rt,imm16",其功能如下。

若 R[Rs]>R[Rt]则 PC=PC+4+imm16×4 否则 PC=PC+4。

假定 ALU 能产生 ZF(零)、NF(符号)和 VF(溢出)三个标志的输出,请在图 6.33 所示的多周期数据通路中增加实现 Bgt 指令的数据通路以及相应的控制信号,并对表 6.10 进行相应的修改,写出该指令对应的微程序(提示:只有一条微指令,可参照 beq 指令实现)。

15. 假定微程序控制器的控存容量为 1024×48 位,微程序可在整个控存内实现转移,反映所有指令执行状态转换的有限状态机中有 4 个分支点,采用水平型微指令格式,并使用断定法确定下条微地址,即微地址由专门的下地址字段确定。请设计微指令的格式,说明各字段的含义和位数,并对转移控制字段进行编码。

16. 对于多周期 CPU 的异常和中断处理,回答以下问题。

(1) 对于除数为 0、溢出、无效指令操作码、无效指令地址、无效数据地址、缺页、访问越权和外部中断,CPU 在哪些指令的哪个时钟周期能分别检测到这些异常或中断?

(2) 在检测到某个异常或中断后,CPU 通常要完成哪些工作?简要说明 CPU 如何完成这些工作?

(3) TLB 缺失和 cache 缺失各在哪个指令的哪个时钟周期被检测到?如果检测到发生了 TLB 缺失和 cache 缺失,那么,CPU 各要完成哪些工作?简要说明 CPU 如何完成这些工作(提示:TLB 缺失可以有软件和硬件两种处理方式)?

第 7 章 指令流水线

第 6 章介绍的单周期处理器和多周期处理器的指令执行都是采用串行方式。串行方式下，CPU 总是在执行完一条指令后才取出下条指令执行。显然，这种串行方式没有充分利用执行部件的并行性，因而指令执行效率低。与现实生活中的许多情况一样，指令的执行也可以采用流水线方式，将多条指令的执行相互重叠起来，以提高 CPU 执行指令的效率。

本章主要介绍指令流水线的基本概念、流水线数据通路和控制器的实现、指令流水线中各种冲突（冒险）现象及其解决方法，并简要介绍一些高级流水线技术。

7.1 流水线概述

7.1.1 流水线的执行效率

一条指令的执行过程可被分成若干个阶段，每个阶段由相应的功能部件完成。如果将各阶段看成相应的流水段，则指令的执行过程就构成了一条指令流水线。例如，假定一条指令流水线由如下 5 个流水段组成。

取指令（IF）：从 cache 或主存取指令。

指令译码（ID）：产生指令执行所需的控制信号。

取操作数（OF）：读取存储器操作数或寄存器操作数。

执行（EX）：对操作数完成指定操作。

写回（WB）：将操作数写回存储器或寄存器。

进入流水线的指令流，由于后一条指令的第 i 步与前一条指令的第 $i+1$ 步同时进行，从而使一串指令总的完成时间大为缩短。如图 7.1 所示，在理想状态下，完成 4 条指令的执行只用了 8 个时钟周期，若是非流水线的串行执行处理，则需要 20 个时钟周期。

从图 7.1 可看出，理想情况下，每个时钟都有一条指令进入流水线；每个时钟周期都有一条指令完成；每条指令的时钟周期数（即 CPI）都为 1。

为了更加清楚地了解流水线的执行效率，下面用一个例子来比较流水线处理器和单周期处理器的指令执行情况。

对于第 6 章给定的具有 11 条指令的单周期处理器（其数据通路见图 6.24），考虑最复杂的 lw 指令的执行情况。假定 lw 指令的 5 个阶段所用的操作时间分别如下。(1)取指：200ps；(2)寄存器读：50ps；(3)ALU 操作：100ps；(4)存储器读：200ps；(5)寄存器写：

50ps。不考虑控制单元、PC 访问、信号传递等延迟，lw 指令的总执行时间为 600ps。

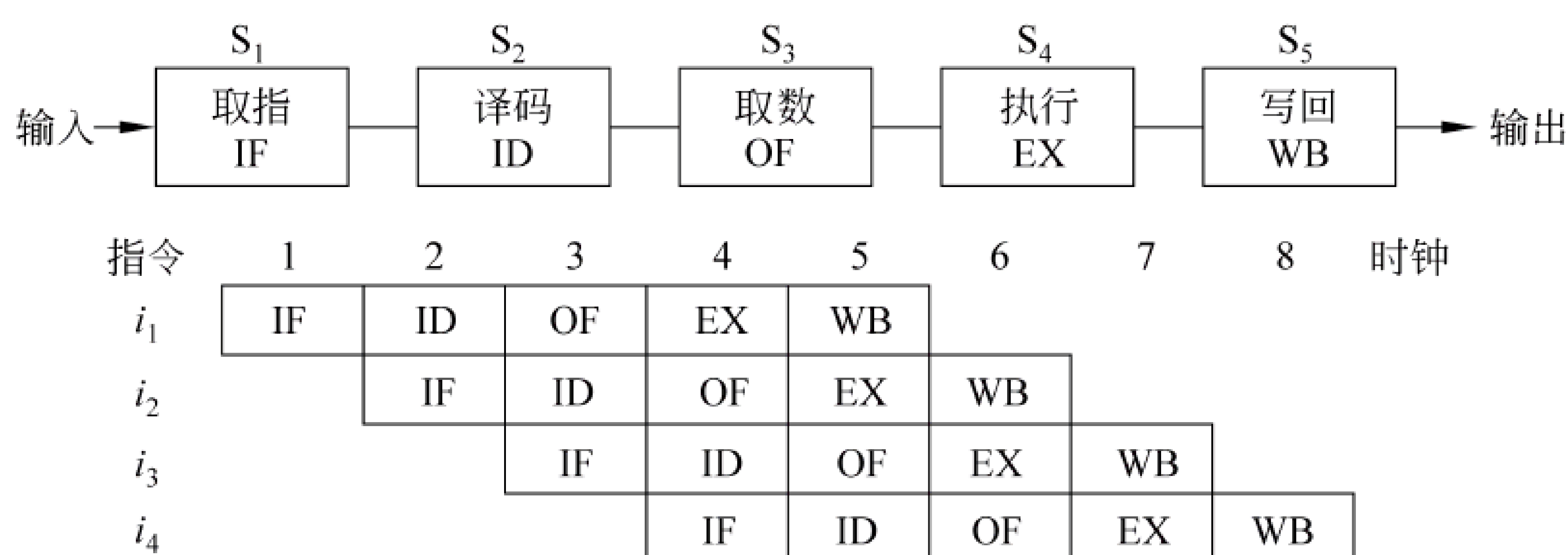


图 7.1 一个 5 段指令流水线

流水线设计的原则是：指令流水段个数以最复杂指令所用的功能段个数为准；流水段的长度以最复杂的操作所用时间为准。考虑实现第 6 章所述的同样 11 条指令的流水线处理器，按照以上流水线设计原则，则该流水线处理器共有 5 个流水段，每个流水段的长度为 200ps，所以，每条指令的执行时间为 1ns，反而比串行执行时增加了 400ps，因此流水线方式并不能缩短一条指令的执行时间。但是，对于整个程序来说，流水线方式可以大大增加指令执行的吞吐率。若流水段数为 M，每个流水段的执行时间为 T，则 N 条指令的执行总时间为 $(M+N-1)\times T$。例如，对于上述 11 条指令的 5 段流水线的例子，假定某程序有 N 条指令，在不考虑任何其他额外开销和冲突的情况下，单周期处理器所用的时间为 $N\times 600$ps，而流水线处理器所用时间为 $(4+N)\times 200$ps。当 N 很大时，流水线方式是串行执行方式的 3 倍。显然，如果每个功能段划分均匀，使得执行时间大致相等的话，提高倍数应为 5，即为流水段的个数。

7.1.2 适合流水线的指令集特征

具有什么特征的指令集有利于实现指令流水线呢？

首先，指令长度应尽量一致。这样，有利于简化取指令和指令译码操作，例如，MIPS 指令都是 32 位，每条指令占 4 个存储单元，因此，每次取指令都是读取 4 个单元，且下址计算也方便，只要 PC+4 即可；而 80x86 指令从 1 字节到 15 字节不等，使取指令部件极其复杂，取指令所用时间也长短不一，而且也不利于指令译码。

其次，指令格式应尽量规整，尽量保证源寄存器的位置相同。这样，有利于在指令未知时就可取寄存器操作数。例如，MIPS 指令格式中，源操作数寄存器 Rs 和 Rt 的位置总是分别固定在 IR＜25：21＞和 IR＜20：16＞，在指令译码的同时就可读取寄存器 Rs 和 Rt 中的内容。若源操作数寄存器的位置随指令不同而不同，则必须先译码后才能确定指令中各寄存器编号的位置，因此，从寄存器取数的工作就不能提前到和译码操作同时进行。

第三，采用装入/存储型指令风格，可以保证除 Load/Store 指令外的其他指令（如运算指令）都不访问存储器，这样，可把 Load/Store 指令的地址计算和运算指令的执行步骤规整在同一个周期中，因此，有利于减少操作步骤，规整流水线。像在 Intel IA-32 之类的非装入/存储型体系结构中，运算类指令的操作数可以是存储器数据，这样，在指令执行过程中，需要有存储器地址计算、存储器访问和运算等，因而这类指令的执行要多出一些功能段，与简单指令的功能段划分相差很大，不利于流水线的规划。

此外，数据和指令在存储器中要“对齐”存放。这样，有利于减少访存次数，使所需数据在一个流水段内就能从存储器中得到。

总之，规整、简单和一致等特性有利于指令的流水线执行。

7.2 流水线处理器的实现

为便于和单周期处理器、多周期处理器比较，假定后面介绍的流水线处理器的实现目标也是第6章提出的11条MIPS指令。以下主要介绍支持该11条指令的流水线数据通路和控制器的实现。

7.2.1 每条指令的流水段分析

指令流水线设计的第一步是要对每条指令的执行过程进行分析，以确定流水线每个功能段的功能和执行时间。

每条指令前两个功能段都一样，它们的功能如下。Ifetch：取指并计算PC+4；Reg/Dec：寄存器取数并译码。后面的功能段随各指令功能的不同而不同。

1. R-型指令功能段划分

R-型指令都涉及到在ALU中对Rs和Rt内容进行运算，最终把ALU的运算结果送目的寄存器Rd。像add和sub等指令还要判断结果是否溢出，只有不溢出时才写结果到Rd，否则转异常处理程序执行。

根据R-型指令的功能，对照第6章多周期数据通路设计，很容易给出R-型指令的功能段划分。如图7.2所示，在Ifetch和Reg/Dec两个公共功能段后，其余的是：Exec功能段用于在ALU中计算；Write功能段用于将ALU中的计算结果写回寄存器。

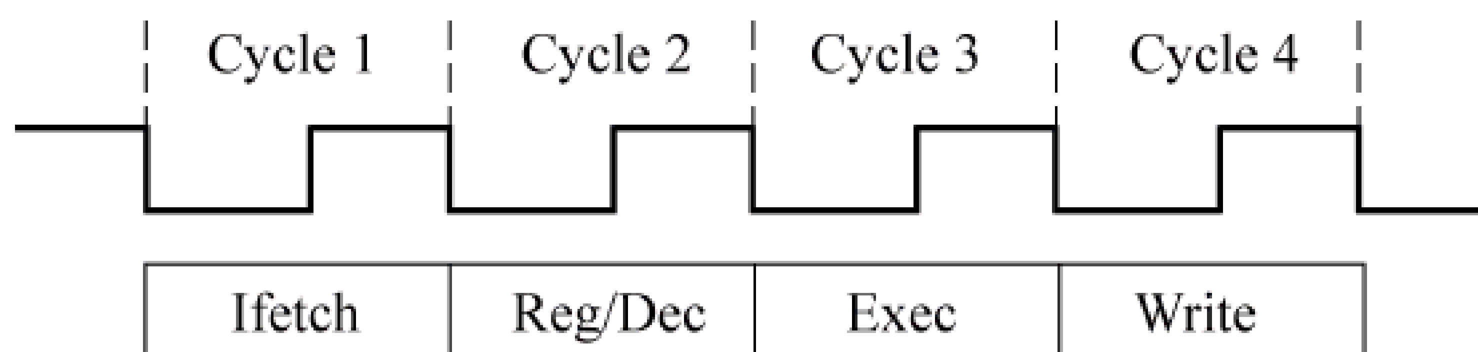


图7.2 R-型指令的功能段划分

2. I-型运算类指令功能段划分

I-型带立即数的运算类指令都涉及到对16位立即数进行符号扩展或零扩展，然后和Rs的内容进行运算，最终把ALU的运算结果送目的寄存器Rt。显然，I-型运算类指令的功能段划分与R-型指令相同。

3. lw指令功能段划分

lw指令的功能为R[Rt]←M[R[Rs]+SignExt(imm16)]。其功能段的划分如图7.3所示，除公共的两个功能段外，其余的是：Exec功能段用于在ALU中计算地址；Mem功能段用于从存储器中读数据；Write功能段用于将数据写入寄存器。

4. sw指令功能段划分

sw指令的功能为M[R[Rs]+SignExt(imm16)]←R[Rt]，即把寄存器内容写入存储器中，与lw指令相比，少了一步写寄存器的工作，其功能段划分如图7.4所示。其中，后面两个功能段的功能是，Exec用于在ALU中计算地址；Mem用于将数据写入存储器中。

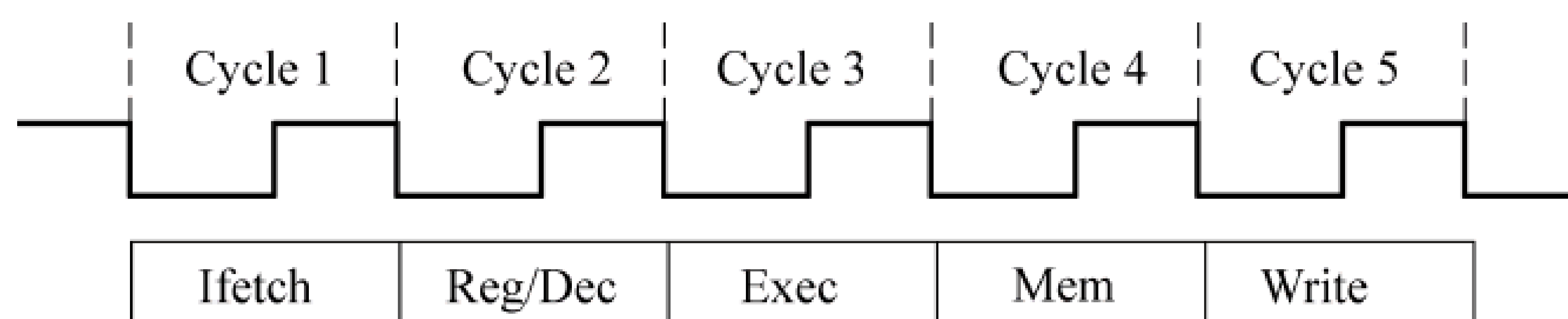


图 7.3　lw 指令的功能段划分

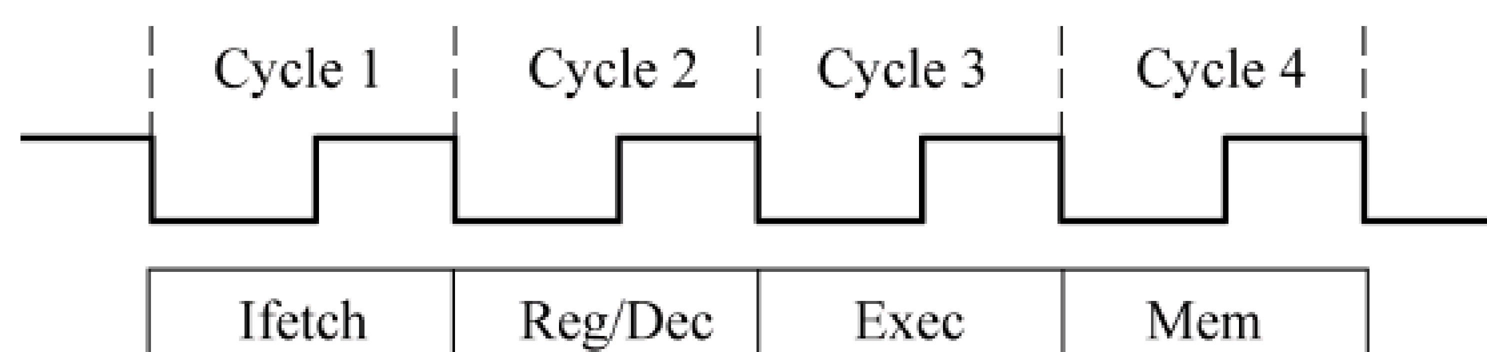


图 7.4　sw 指令的功能段划分

5. beq 指令功能段划分

beq 指令的功能为 if (R[Rs] = R[Rt]) then PC←PC+4+(SignExt(imm16)×4) else PC←PC+4。除了前面两个公共功能段外，其后各功能段可以划分为：Exec 用于在 ALU 中做减法以比较是否相等，同时用一个加法器计算转移地址；WrPC 功能段用于在比较相等的情况下将转移目标地址写到 PC 中。因为写入 PC 的操作(WrPC)比存储器访问操作(Mem)的时间短，所以，可以将功能段 WrPC 向功能段 Mem 靠，即最后的功能段用 Mem 表示。因此，beq 的功能段划分类似于 sw 指令，如图 7.4 所示。

6. j 指令功能段划分

j 指令是无条件转移指令，其功能是直接将目标地址送 PC 中。所以，其功能段的划分很简单，除了两个公共的功能段外，就只有一个功能段 WrPC，其操作时间比 Exec 段时间短。

从以上对各指令功能段的分析可看出，最复杂的是 lw 指令，它有 5 个功能段，其他指令都可以通过加入"空"功能段来向 lw 指令靠齐。

在插入"空"段时，应遵循以下两个原则：(1)每个功能部件每条指令只能用一次(如寄存器写口不能用两次或以上)；(2)每个功能部件必须在相同的阶段被使用(如寄存器写口总是在第 5 阶段被使用)。

因此，R-型指令、I-型运算类指令需在 Write 之前加一个空的"Mem"段，使得其 Write 段和 lw 指令的 Write 对齐，都在第 5 段；sw 指令和 beq 指令在第 4 个功能段后加一个空的"Write"段；j 指令则在后面添加两个空段"Mem"和"Write"。这样，所有指令都有 5 个功能段。因此，该处理器的指令流水线可以设计成 5 个流水段。

7.2.2　流水线数据通路的设计

根据对 11 条指令的分析，可以得到执行这 11 条指令的 5 段流水线数据通路基本框架，如图 7.5 所示。

在图 7.5 所示的流水线数据通路中，每条指令的执行都经历 5 个流水段：IF、ID、Ex、Mem 和 Wr，每个流水段都在不同的功能部件中执行。流水段之间有一个流水段寄存器，例如，IF/ID 寄存器是介于 IF 段和 ID 段之间的寄存器。每个流水段寄存器用来存放从当前流水段传到后面所有流水段的信息。因为每个段间传递的信息不一样，所以各流水段寄存器的长度也不一样。

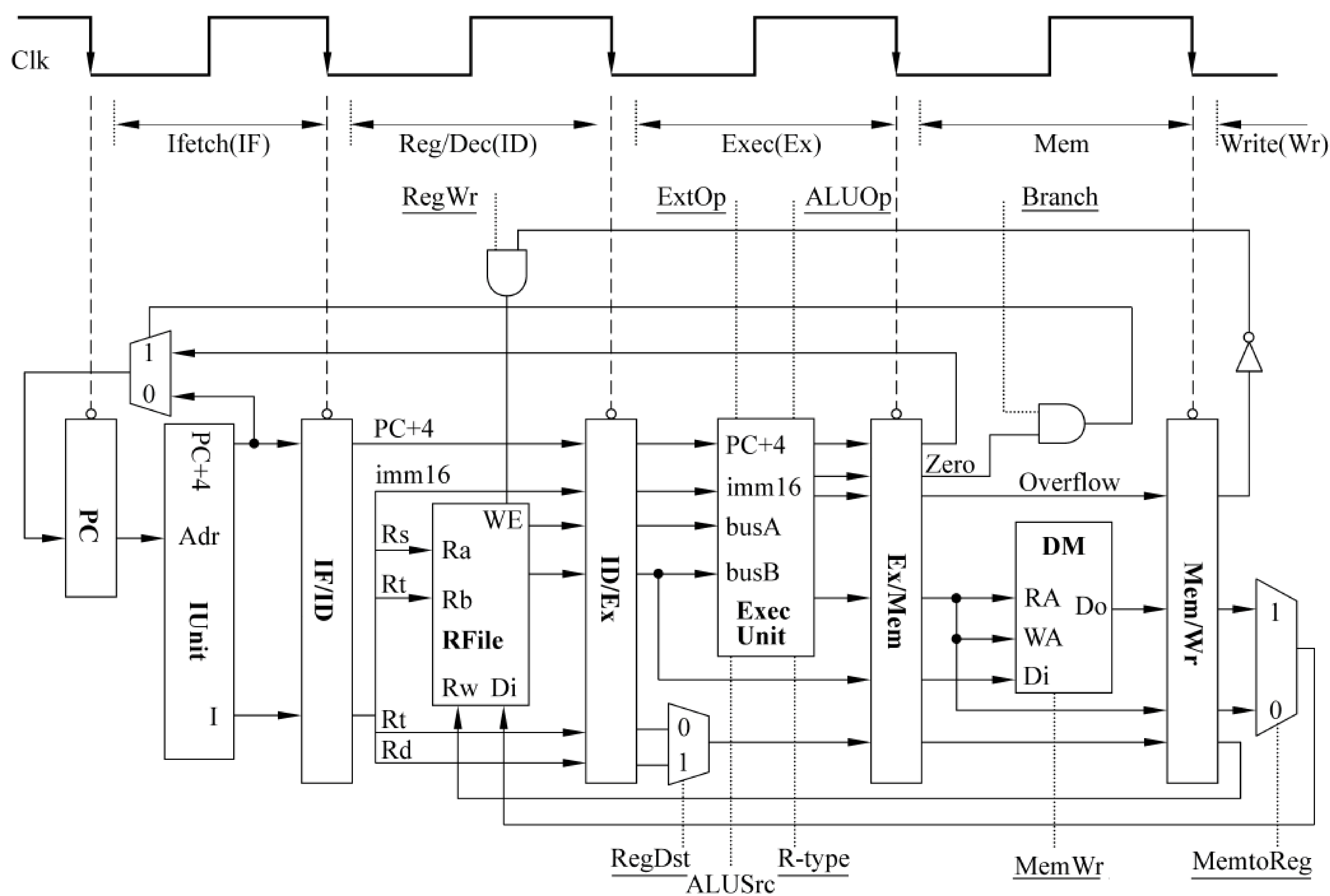


图 7.5　5 段流水线数据通路基本框架

图 7.5 中给出了理想情况下数据通路用到的所有控制信号,用点虚线连到所控制的功能部件。可以看出,PC 和各个流水段寄存器都没有写使能信号。这是因为每个时钟都会改变 PC 的值,所以 PC 不需要写使能控制信号;每个流水段寄存器在每个时钟都会写入一次,因此,流水段寄存器也不需要写使能控制信号。此外,前两个流水段的功能每条指令都相同,是公共流水段,因此,也不需控制信号。其余段的控制信号如下。

Exec 段的控制信号有以下 5 个。

　　ExtOp(扩展器操作):1—符号扩展;0—零扩展。

　　ALUSrc(ALU 的 B 口来源):1—来源于扩展器;0—来源于 busB。

　　ALUOp(用于辅助局部 ALU 控制逻辑来决定 ALUCtr 的操作信号):三位编码。

　　RegDst(指定目的寄存器):1—Rd;0—Rt。

　　R-Type(区分是否为 R-型指令):1—R-型指令;0—非 R-型指令。

Mem 段的控制信号有以下两个。

　　MemWr(数据存储器 DM 的写信号):sw 指令时为 1,其他指令为 0。

　　Branch(是否为分支指令):分支指令时为 1,其他指令为 0。

Wr 段的控制信号有以下两个。

　　MemtoReg(寄存器的写入源):1—DM 输出;0—ALU 输出。

　　RegWr(寄存器堆写信号):结果写寄存器的指令都为 1,其他指令为 0。

以下分别介绍各流水段的功能、功能部件、保存到流水段寄存器的信息和控制信号取值。

1. Ifetch(IF)段

IF 流水段的功能是:将 PC 的值作为地址到指令存储器 IM(Instruction Memory)中取

指令，并计算 PC+4，送 PC 输入端。这些功能由取指部件(IUnit)来完成，其具体实现如图 7.6 所示。

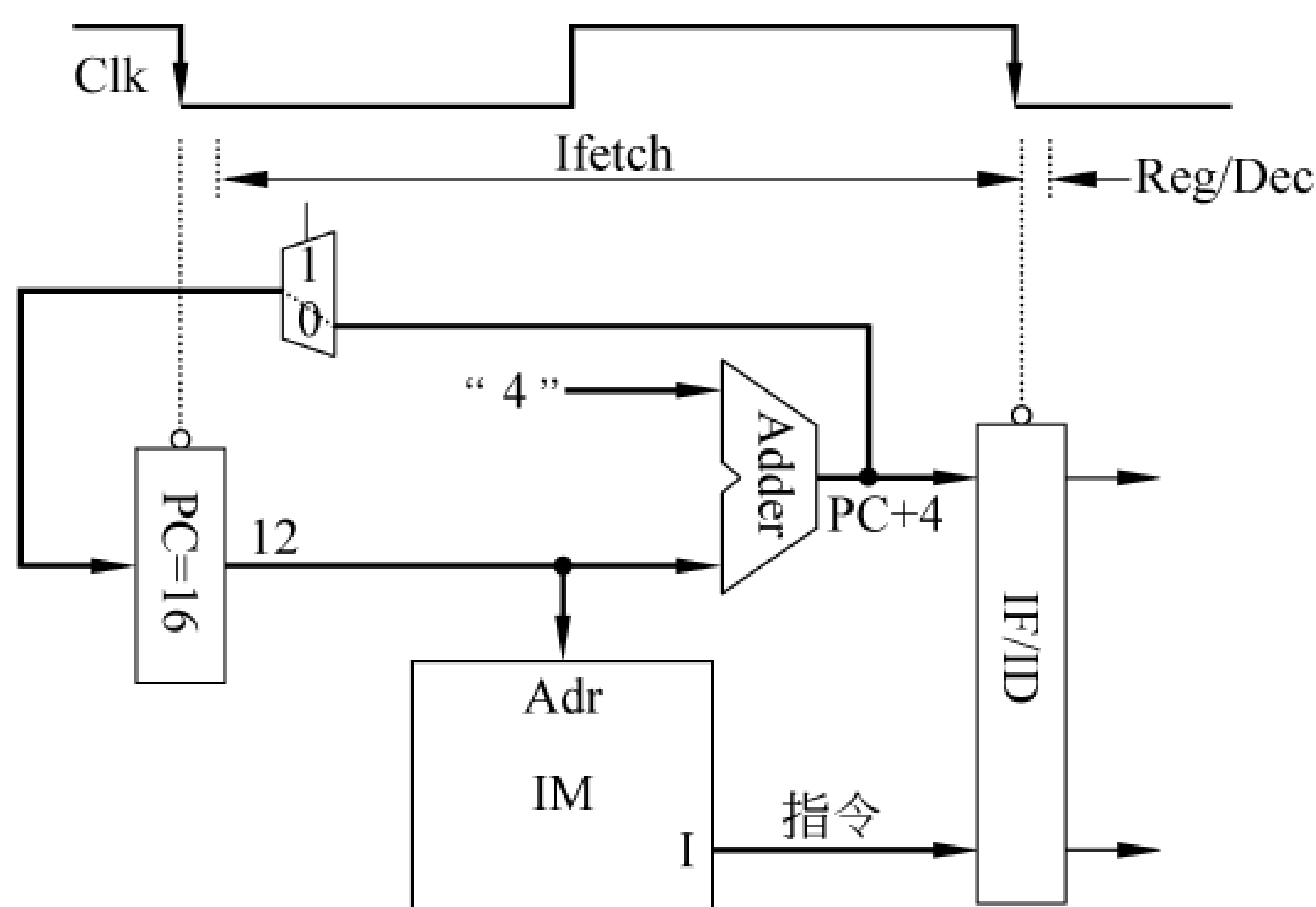


图 7.6 取指令部件 IUnit 的内部实现

假定当前指令地址为 12，则当时钟 Clk 的下降沿到来时，在 PC 输入端的值 12 经过"Clk-to-Q"时延后，被送到 IM 的地址输入端 Adr，并同时送加法器。在 IM 中经过一个存取时间后，指令被送到 IM 的输出端。在加法器中 PC 与"4"相加后送到一个多路选择器，若是顺序执行，则下个时钟到来时 PC 为 16。但指令不总是顺序执行，当执行到分支指令或无条件转移指令时，PC 的值可能被修改，因此 PC 的输入来自一个多路选择器。当需要转移时，可控制选择转移目标地址送 PC。

IF 段执行的结果被送到 IF/ID 寄存器的输入端，下个时钟到来时，在 IF/ID 寄存器输入端的信息开始送到 ID 段继续被处理。那么，IF/ID 寄存器中需要保存的结果有哪些呢？显然，从 IM 中取出的指令要被继续处理，因而，需要保存在 IF/ID 寄存器中；此外，如果当前指令是分支指令的话，则 PC+4 的值在后面的流水段中需要用来计算相对转移地址，所以，也需要保存在 IF/ID 寄存器中。

该段唯一的控制点是多路选择器的控制端，从图 7.5 看出，多路选择器的控制端由在 Mem 段产生的 Branch 信号和 Zero 标志来控制，显然，Branch 信号只有在对分支指令 beq 译码后才取值为 1，所以，在 IF 阶段 Branch=0，因而，此时，多路选择器的输出为 PC+4。在执行 beq 指令时，在 Mem 段将得到转移目标地址，此时，若 Zero=1 则将转移目标地址选择送到 PC 的输入端。

2. Reg/Dec(ID)段

ID 流水段的功能是：根据指令中的 Rs 和 Rt 到寄存器堆中取出相应寄存器的值，同时对指令中的操作码 op 字段进行译码，生成相应的控制信号。寄存器堆可看成是寄存器读口和寄存器写口两个功能部件。ID 段的功能由寄存器读口和控制器完成，如图 7.5 所示。有关流水线处理器的控制器的实现在 7.2.3 节介绍。

ID 段执行的结果被送到 ID/Ex 寄存器的输入端，下个时钟到来时，在 ID/Ex 寄存器输入端的信息开始送到 Ex 段继续被处理。这些信息包括 Reg[Rs]、Reg[Rt]、Rt、Rd、imm16、func、PC+4 等。因为指令中需要的信息已被保存，所以指令就不再需要保存在 ID/Ex 中。

3. Exec(Ex)段

Ex 段的功能由具体指令确定,不同指令经 ID 段译码后得到不同的控制信号,用来控制执行部件进行不同的操作。图 7.7 是执行部件(Exec Unit)的示意图。

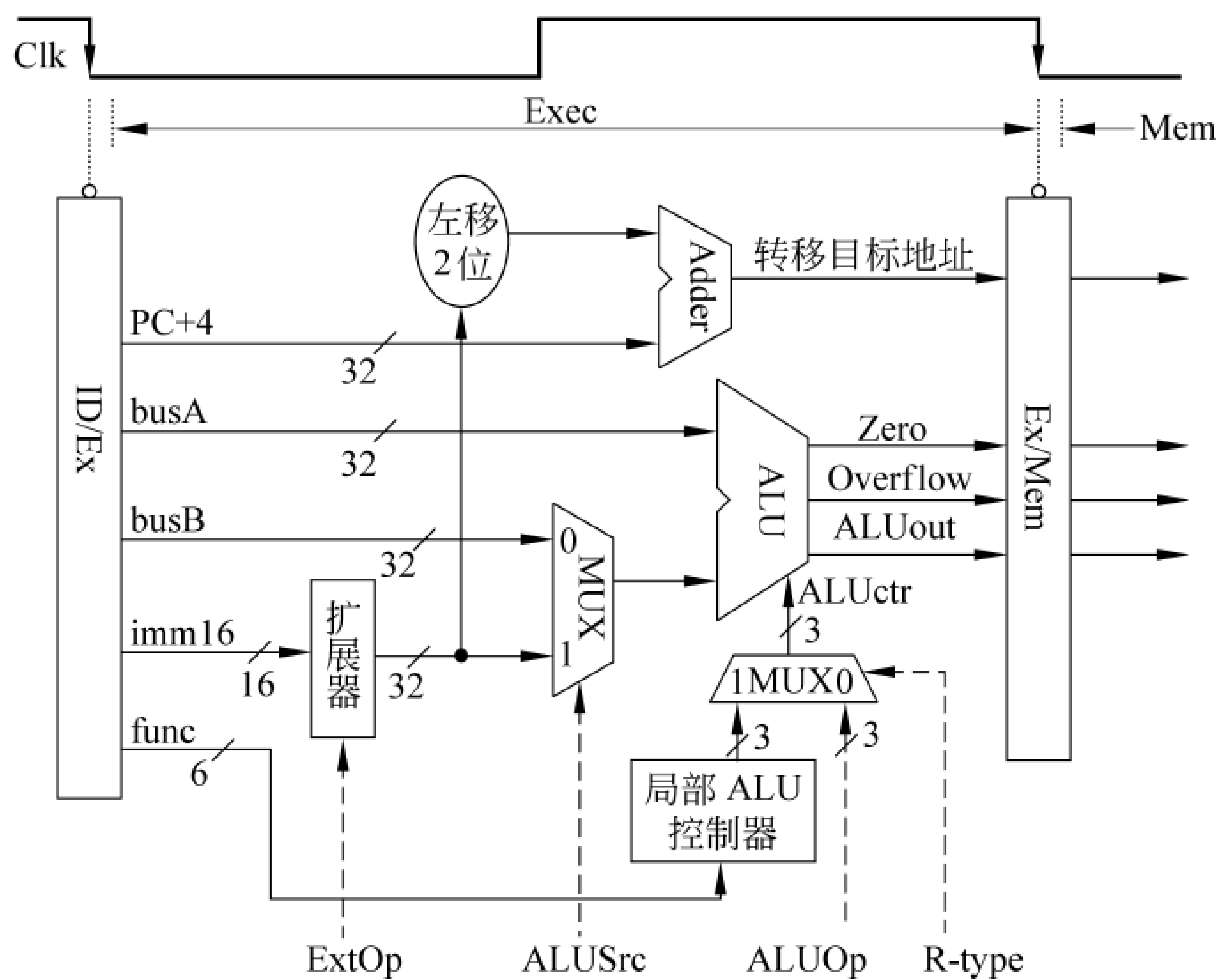


图 7.7　执行部件 Exec Unit 的内部实现

在 Ex 流水段中,各类指令的功能分别如下。

(1) R-型指令。在 ALU 中对 busA 和 busB 传递来的数据执行相应的运算,运算结果送 ALUout,并产生相应的标志信息 Zero 和 Overflow。

(2) I-型运算类指令。在 ALU 中对立即数扩展后的数据与 busA 传递来的数据执行相应的运算,运算结果送 ALUout,并产生相应的标志信息 Zero 和 Overflow。

(3) lw/sw 指令。在 ALU 中对立即数符号扩展后的数据与 busA 传递来的数据执行加法运算,得到存储器地址送 ALUout。

(4) beq 指令。在 ALU 中对 busA 和 busB 传递来的数据执行减法运算,得到 Zero 标志;同时,对立即数进行符号扩展后,在加法器中与“PC+4”相加,得到转移目标地址。

(5) j 指令。生成转移目标地址,即 PC<31:28>||target<25:0>。

根据每类指令的功能,综合考虑图 7.7 和图 7.5,得到每条指令的执行流程及其控制信号取值如下。

(1) R-型指令的执行

11 条目标指令中的 add、sub、subu、slt 和 sltu 都是 R-型指令,它们在 ALU 中由 ALUctr 控制分别执行 add、sub、subu、slt 和 sltu 运算,ALUctr 操作控制信号由局部 ALU 控制器根据 func 字段产生。R-型指令的目的寄存器是 Rd,ALU 的操作数来自于 busA 和 busB,不需要扩展操作。最终,将 ALU 得到的结果,包括 Overflow 标志和 Zero 标志一起被送到 Ex/Mem 寄存器的输入端。综上所述,得到控制信号的取值为 RegDst=1,ExtOp=x,ALUSrc=0,ALUop=xxx,R-type=1。

(2) I-型运算类指令的执行

11 条目标指令中的 ori 和 addiu 是 I-型运算类指令，它们在 ALU 中由 ALUctr 控制分别执行 or 和 addu 运算，ALUctr 操作控制信号由主控制器根据指令操作码 op 字段产生。I-型运算类指令的目的寄存器是 Rt，ALU 的操作数来自于 busA 和扩展器的输出，逻辑运算进行零扩展，而算术运算则为符号扩展。与 R-型指令一样，最终将 ALU 中得到的结果，包括 Overflow 标志和 Zero 标志一起被送到 Ex/Mem 寄存器的输入端。综上所述可知，ori 指令的控制信号取值为 RegDst＝0，ExtOp＝0，ALUSrc＝1，ALUop＝or，R-type＝0。addiu 指令的控制信号取值为 RegDst＝0，ExtOp＝1，ALUSrc＝1，ALUop＝addu，R-type＝0。

(3) lw 指令的执行

首先要在 ALU 中进行地址计算，ALU 的操作数来自于 busA 和扩展器输出，采用符号扩展，在 ALU 中由 ALUctr 控制执行 addu 运算，目的寄存器是 Rt。最终在 ALU 中得到的存储器地址被送到 Ex/Mem 寄存器的输入端。综上所述，得到控制信号的取值为 RegDst＝0，ExtOp＝1，ALUSrc＝1，ALUop＝addu，R-type＝0。

(4) sw 指令的执行

同 lw 指令一样，需要进行存储器地址计算并送下一级流水线寄存器。因为该指令不会写结果到寄存器，所以，RegDst 的取值可任意，不会影响结果。综上所述，得到控制信号的取值为 RegDst＝x，ExtOp＝1，ALUSrc＝1，ALUop＝addu，R-type＝0。

(5) beq 指令的执行

beq 指令需要比较寄存器 Rs 和 Rt 的值，通过在 ALU 中做减法生成 Zero 标志来实现比较。因此，ALU 两个操作数来源是 busA 和 busB，ALUctr 操作控制信号为 subu；同时，将 imm16 送到扩展器，然后在 ExtOp 的控制下进行符号扩展，扩展结果左移两位，再和 PC＋4 相加，生成相对转移目标地址。执行阶段生成的 Zero 标志和转移目标地址被送到 Ex/Mem 寄存器的输入端。因为不改变任何寄存器的值，所以控制信号 RegDst 的值任意。综上所述，得到控制信号的取值为 RegDst＝x，ExtOp＝1，ALUSrc＝0，ALUop＝subu，R-type＝0。

(6) j 指令的执行

只要形成一个 32 位的转移目标地址(PC＜31：28＞||target＜25：0＞)即可。所以不需要执行任何运算，只要将相应的信号线串联起来形成 32 位地址送到 Ex/Mem 寄存器的输入端。显然，控制信号的取值为 RegDst＝x，ExtOp＝x，ALUSrc＝x，ALUop＝xxx，R-type＝x。有关 j 指令的数据通路没有画出，留待习题中完成，可参照 beq 指令来实现。

4. Mem 段

Mem 流水段的功能也由具体指令确定。从图 7.5 知，这个流水段有 Branch 和 MemWr 两个控制信号。各条指令在 Mem 段的执行流程和控制信号取值如下。

(1) 若是 R-型指令或 I-型运算类指令，则在 Mem 段是“空”操作，只要把相应信息继续传递到下一个流水段即可。控制信号取值为 Branch＝0，MemWr＝0。

(2) 若是 lw 指令，则进行取数操作。在 Ex 段得到的地址被送到 DM 的读地址端 RA，经过一段存取时间，数据从 DM 的输出端 Do 送到 Mem/Wr 寄存器的输入端。控制信号取值为 Branch＝0，MemWr＝0。

(3) 若是 sw 指令，则进行存数操作。在 Ex 段得到的地址被送到 DM 的写地址端

WA，同时把 Ex/Mem 寄存器送来的要存的数据 Reg[Rt]送 DM 的输入端 Di，经过一段存取时间后，数据被存入 DM 中。控制信号取值为 Branch=0，MemWr=1。

(4) 若是 beq 或 j 指令，则将 Ex 段生成的转移目标地址更新到 PC 中，图 7.5 和图 7.7 中画出了 beq 指令相关的数据通路。若是 beq 指令，则 Branch=1，MemWr=0。此时，若在 Ex 段生成的 Zero 为 1，则会控制 PC 输入端的多路选择器选择将转移目标地址送 PC 的输入端。

5. Wr 段

如图 7.5 所示，寄存器写口是 Wr 段的主要功能部件，寄存器堆 RFile 的写地址端口 Rw 来源于 Mem/Wr 寄存器的目的寄存器输出，写数据端口 Di 来源于一个多路选择器的输出，写使能信号 WE 由溢出标志 Overflow 和控制信号 RegWr 共同确定。Wr 流水段的功能也由具体指令确定，各条指令在 Wr 段的执行流程和控制信号取值如下。

(1) 若是 R-型指令或 I-型运算类指令，则选择将 ALU 的输出结果送寄存器堆的输入端 Di，目的寄存器送写地址端 Rw。控制信号取值为 MemtoReg=0，RegWr=1。

(2) 若是 lw 指令，则选择将 DM 读出结果送寄存器堆的输入端 Di，目的寄存器送写地址端 Rw。控制信号取值为 MemtoReg=1，RegWr=1。

(3) 若是 sw、beq 或 j 指令，则任何寄存器的值都不改变，即不能写寄存器。控制信号取值为 MemtoReg=x，RegWr=0。

7.2.3 流水线控制器的设计

从上述分析可以看出，某一时刻每个流水段执行的是不同指令的某个阶段，因而某一时刻每个流水段中的控制信号应该是正在执行指令的对应功能段的控制信号。

如图 7.8 所示，假定有三条指令 lw、ori 和 add 依次在时钟 1、2 和 3 开始进入流水线执行，则流水线中控制信号的传递情况如下：第 2 时钟对 lw 指令译码，产生出的控制信号在下个时钟(第 3 时钟)送到 Exec 段，在第 4 时钟送到 Mem 段，在第 5 时钟送到 Write 段；在第 3 时钟对 ori 指令译码，产生出的控制信号在第 4 时钟送 Exec 段，第 5 时钟送 Mem 段，第 6 时钟送 Write 段；……由此可见，在某个时钟期间，不同的流水段受不同指令的控制信号控制，执行不同指令的不同功能段。例如，在第 5 时钟内，Write 段由 lw 指令的信号控制，Mem 段由 ori 的信号控制，Exec 段由 add 的信号控制。

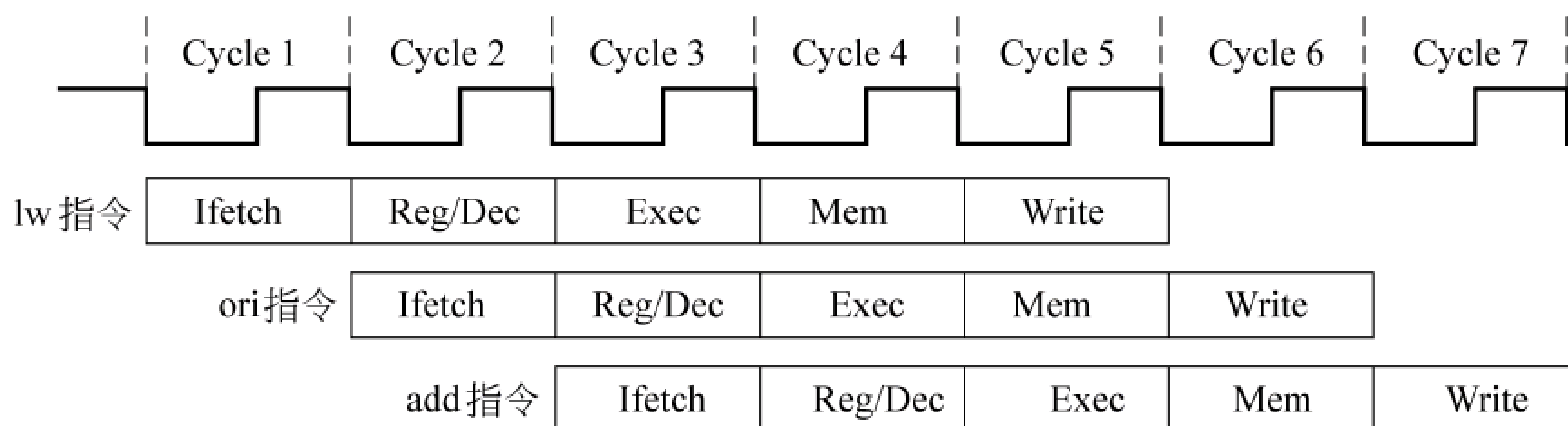


图 7.8　流水线执行情况举例

从上述例子可看出：在 Reg/Dec 阶段由控制器产生指令各流水段的所有控制信号，分别在随后的各个时钟周期内被使用。具体来说，Exec 阶段的信号(RegDst，ExtOp，ALUSrc，ALUop，R-type)在下个周期使用；Mem 阶段的信号(MemWr，Branch)在随后第二个周期使用；Write 阶段的信号(MemtoReg，RegWr)在随后第三个周期使用。因此随后

各流水段寄存器中都要保存相应的控制信号,如图 7.9 所示。

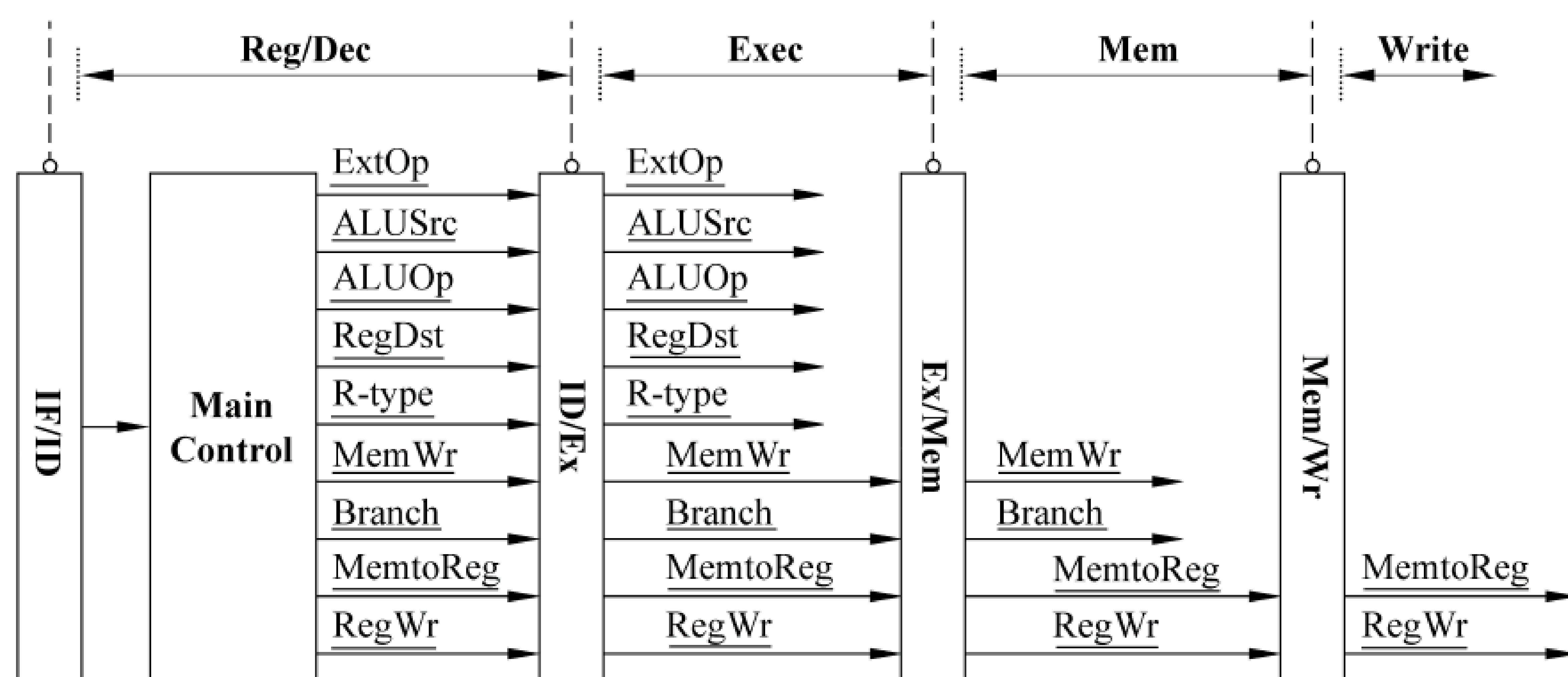


图 7.9　控制信号在流水线中的传递

综上所述,每个流水段寄存器中保存的信息包括两类:一类是后面阶段需要用到的所有数据信息,包括 PC+4、指令、立即数、目的寄存器、ALU 运算结果、标志信息等,它们是前面阶段在数据通路中执行的结果;另一类是前面传递过来的后面各阶段要用到的所有控制信号。

前面第 6 章介绍过单周期处理器和多周期处理器的控制器设计。单周期处理器中,每条指令的控制信号在指令执行期间是不变的;而多周期处理器中,每条指令分多个周期执行,所以控制器的功能采用有限状态机来描述。

因为流水线处理器中控制信号一旦在 ID 段由控制器生成,就不会改变,并和数据信息同步地依次传递到后面的流水段中。显然,这和单周期控制器类似,因而,流水线控制器的设计可以完全按照单周期控制器设计的思路进行。故在此不多赘述。

7.3　流水线冒险及其处理

指令流水线中,可能会遇到一些情况使得流水线无法正确执行后续指令而引起流水线阻塞或停顿(stall),这种现象称为流水线冒险(hazard)。根据导致冒险的原因的不同,有结构冒险、数据冒险和控制冒险三种。以下分别介绍其原因和对策。

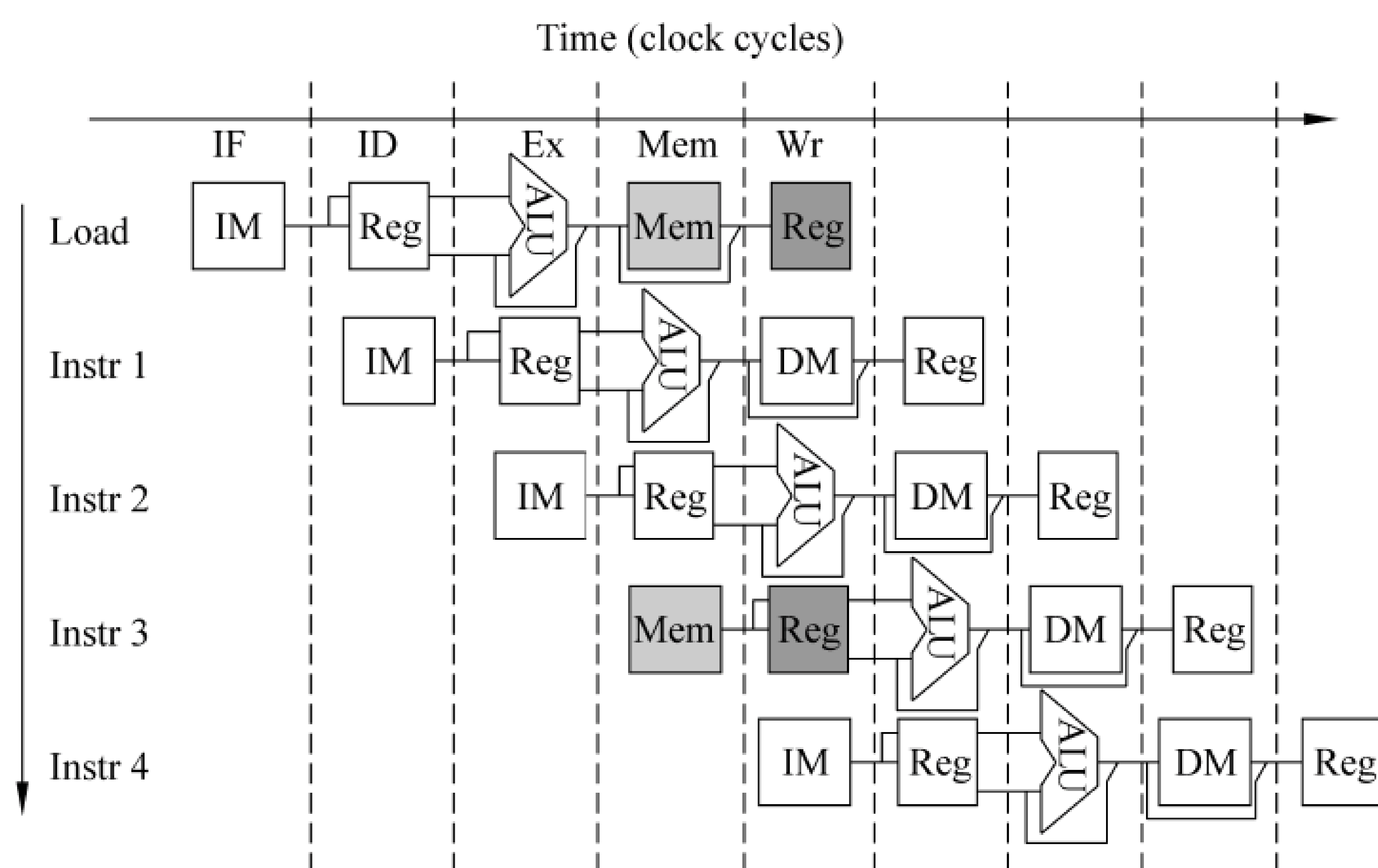
7.3.1　结构冒险

结构冒险(Structural Hazards)也称为硬件资源冲突(Hardware Resource Conflicts),引起结构冒险的原因在于同一个部件同时被不同指令所用,也就是说它是由硬件资源竞争造成的。

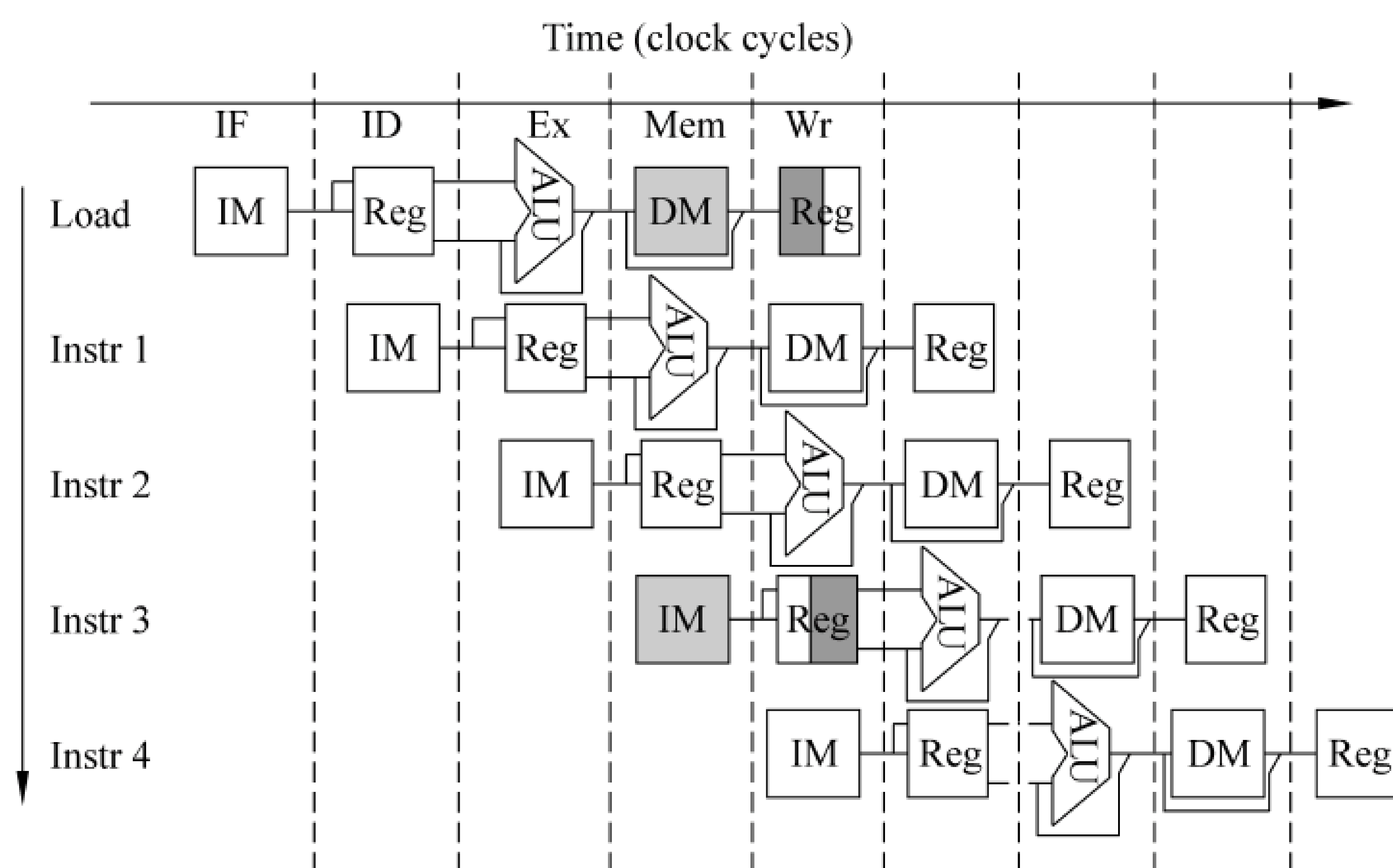
如图 7.10(a)所示,若不区分指令存储器和数据存储器而只用一个存储器的话,则在 Load 指令取数据的同时,随后的指令 3(Instr 3)正好取指令,此时发生访存冲突。同样,如果不对寄存器堆的写口和读口独立设置的话,Load 和随后的指令 3 也会发生寄存器访问冲突。

解决结构冒险的策略有两个方面:(1)通过前面 7.2.1 节提到过的功能段划分原则(一

个部件每条指令只能使用1次，且只能在特定周期使用)，可以避免一部分结构冒险。(2)通过设置多个独立的部件来避免硬件资源冲突。例如，对于寄存器访问冲突，可将寄存器读口和写口独立开来，利用时钟上升沿和下降沿两次触发，使得前半周期使用写口进行寄存器写，后半周期使用读口进行寄存器读；对于存储器访存冲突，可把指令存储器IM和数据存储器DM分开，从而使指令和数据的访问各自独立，这样就不会发生结构冒险，如图7.10(b)所示。事实上，现代计算机都引入了cache机制，而且L1 cache通常采用数据cache和代码cache分离的方式，因而也就避免了结构冒险的发生。



(a) 有寄存器和存储器访问冲突的流水线



(b) 消除了寄存器和存储器访问冲突的流水线

图7.10　结构冒险的例子

7.3.2 数据冒险

数据冒险(Data Hazards)也称为数据相关(Data Dependencies)。引起数据冒险的原因在于后面指令用到前面指令结果时前面指令结果还没产生。如图7.11所示是一个存在数据冒险的流水线例子。

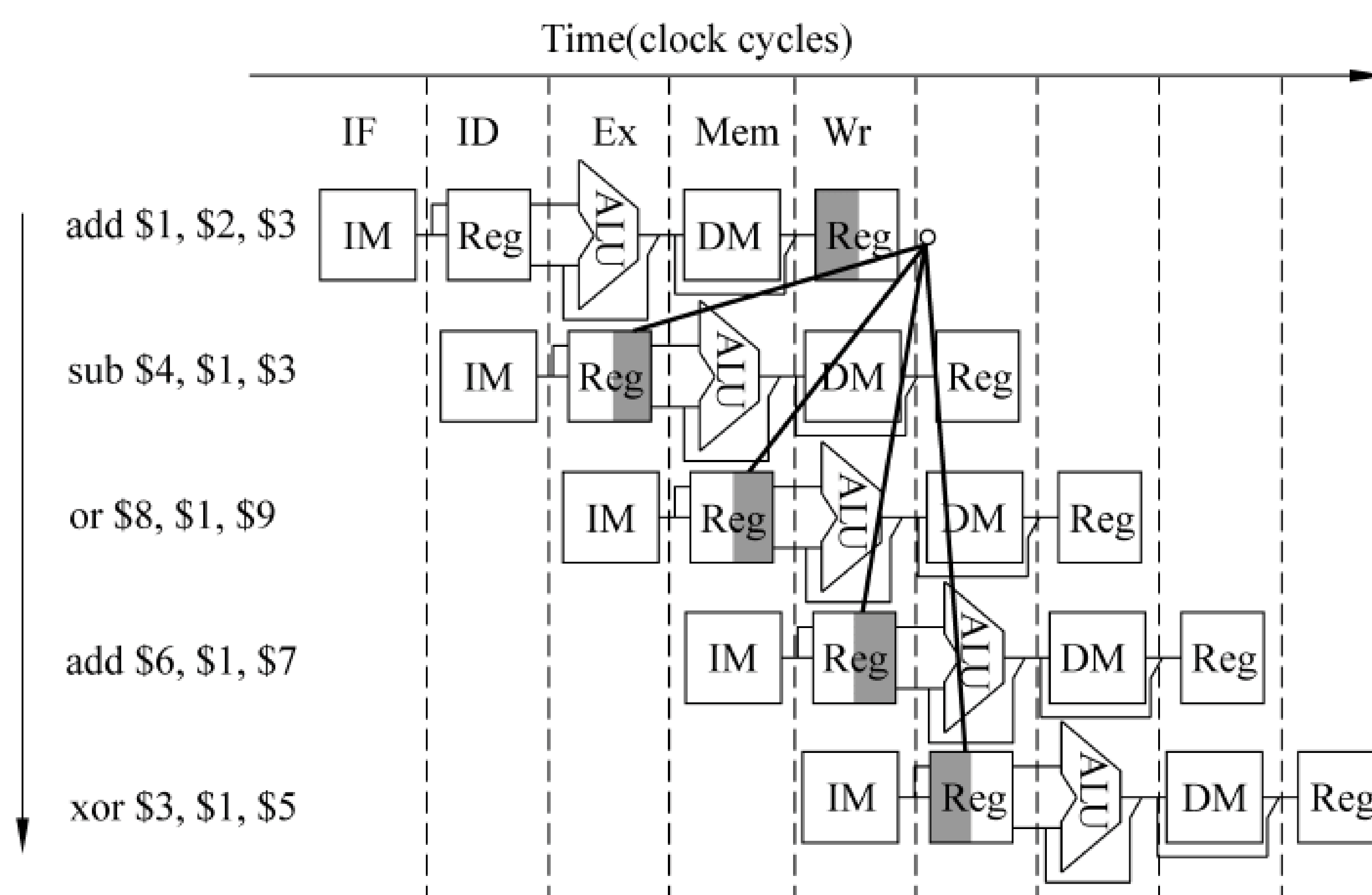


图 7.11　存在数据冒险的流水线例子

在图 7.11 中,第 1 条指令的目的寄存器 \$1 是后面 4 条指令的源寄存器。第 1 条指令在 Wr 阶段结束才将结果写到 \$1 中,而第 2、3、4 条指令分别在第 1 条指令的 Ex、Mem 和 Wr 阶段就要取 \$1 的内容,显然,如果不采取任何措施的话,这几条指令取到的是 \$1 的旧值,只有第 5 条指令 xor 能取到 \$1 的新值。从图 7.11 可看出,所有的数据冒险都是由于前面指令写结果之前后面指令就需要读取而造成的,这种数据冒险称为写后读(Read After Write,RAW)数据冒险。在非“乱序”执行的基本流水线中,所有数据冒险都属于 RAW 数据冒险。

对于 RAW 数据冒险,可以采取以下几个措施。

1. 插入空操作指令

在软件上采取措施,使相关指令延迟执行。最简单的做法是,在编译时预先插入空操作指令 nop。这样做的好处是硬件控制简单,但浪费了指令存储空间和指令执行时间。如图 7.12 所示,共浪费了三条指令的空间和时间。

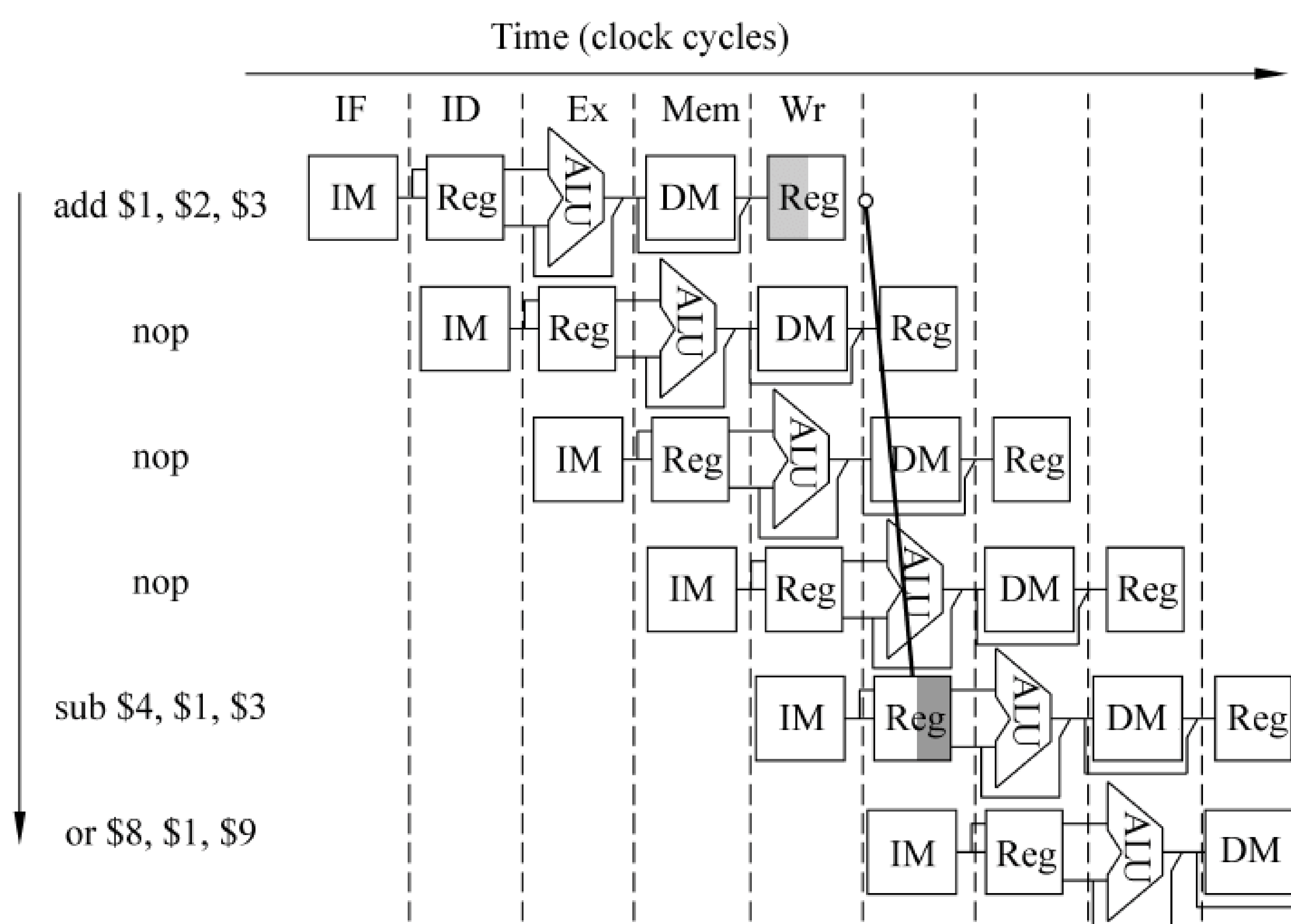


图 7.12　用插入 nop 指令方式解决数据冒险

2. 插入气泡

在硬件上采取措施,使相关指令延迟执行,通过硬件阻塞(stall)方式阻止后续指令执行。这种硬件阻塞的方式称为"插入气泡(bubble)",如图 7.13 所示。

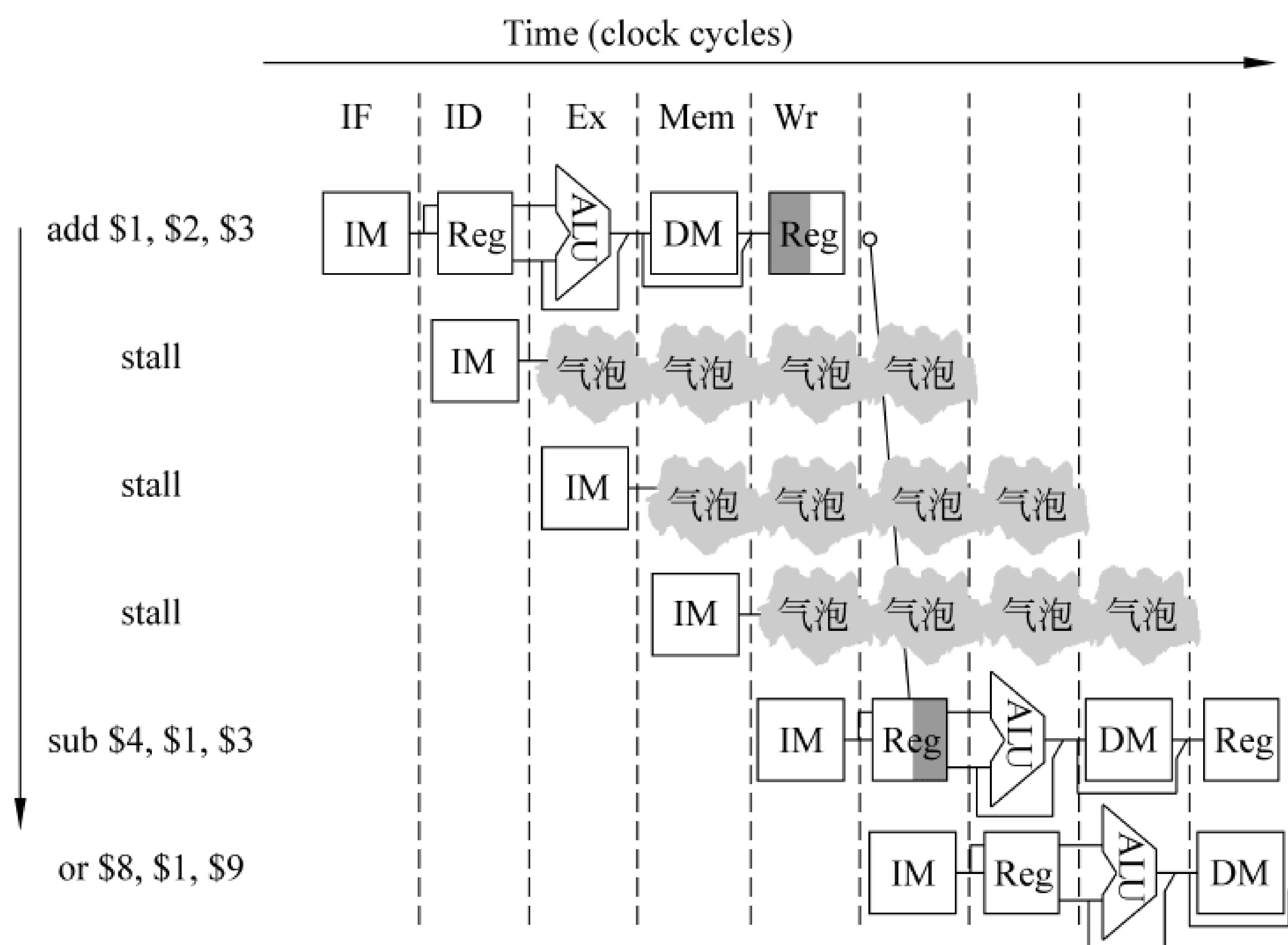


图 7.13 用流水线阻塞方式解决数据冒险

这种方式控制比较复杂,需要修改数据通路。通常要在数据通路中检测哪两条指令发生了相关,以确定是否进行阻塞。阻塞时,可将控制信号清 0 来阻止结果的写入;也可将指令清 0 使后续指令执行空操作;或让 PC 增加一个写使能信号并清 0,使 PC 值不变,从而使当前指令重复执行。这种方式不增加指令条数,但有额外时间开销。

*** 3. 采用转发技术**

将数据通路中生成的中间数据直接转发到 ALU 的输入端。从图 7.11 可看出,第一条指令在 Ex 段结束时已经得到 $1 的新值,被存放在 Ex/Mem 流水段寄存器中,因此,可以直接从流水段寄存器中取出数据送到 ALU 的输入端,这样,在第二条指令执行时 ALU 中用的是 $1 的新值。同样,第三条指令在 ALU 中用到的 $1 也可以直接从 Mem/Wr 流水段寄存器中取,如图 7.14 所示。这种技术称为转发(forwarding)或旁路(bypassing)技术。

对于第一条和第四条指令之间的数据相关问题,可以通过将寄存器写口和读口分别控制在前、后半个时钟周期内操作来解决,使前半周期写入 $1 的值在后半周期马上被读出。

采用转发技术解决数据冒险必须在硬件上进行相应的改动。通过在 ALU 的输入端加多路选择器,使 Ex 段之后的流水段寄存器的值能返送到 ALU 输入端。ALU 的 A 输入端原来只有从 ID/Ex 寄存器来的 busA,B 输入端原来只有从 ID/Ex 寄存器来的 busB 和扩展器的值,采用转发技术后,A 端和 B 端都增加了三个可能的输入,如图 7.15 所示。

图 7.15 中有三个指令序列示例,从这三个示例可看出,增加转发线路后,相邻两条 ALU 运算类指令之间、相隔一条的两个 ALU 运算类指令之间以及相隔一条的 Load 和 ALU 运算类指令之间的数据相关带来的数据冒险问题就都能解决了。

如果是 R-型指令后直接跟 sw 指令的相关性问题,则上述转发线路不能解决。这种情况是,R-型指令的目标寄存器是 sw 指令的源寄存器;sw 指令在译码阶段读取源寄存器的

内容时，上条 R-型指令中最后的正确结果尚未写入目标寄存器。但仍然可以用转发技术来解决，只要采用图 7.15 所示的类似方法，在 DM 的数据输入端 Di 处增加一个多路选择器就可以，即在 sw 指令写存储器时不使用该指令读出的源寄存器值（是一个旧值），而是改用上条 R-型指令执行阶段产生的 ALU 的输出值。

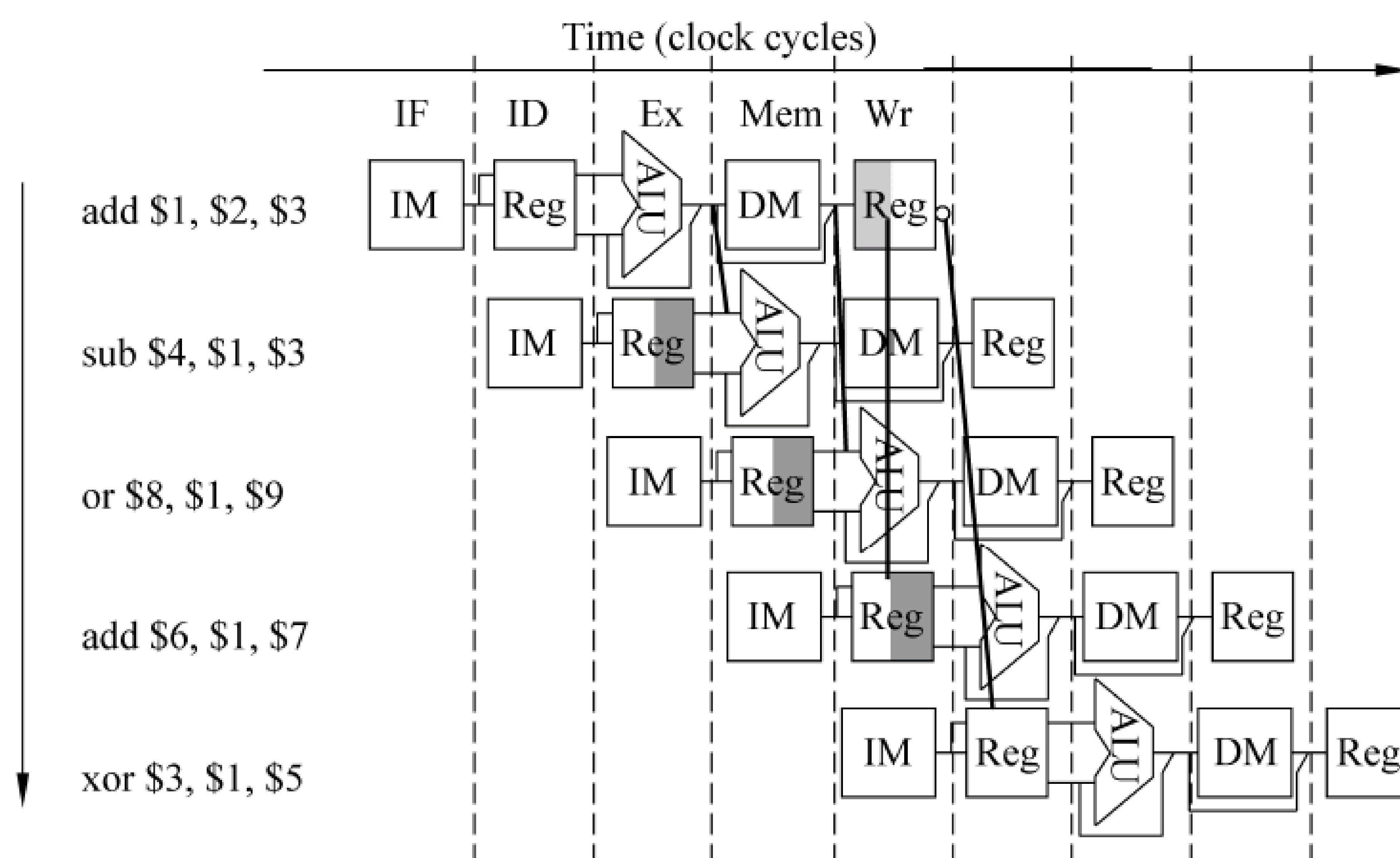


图 7.14　用转发技术解决数据冒险

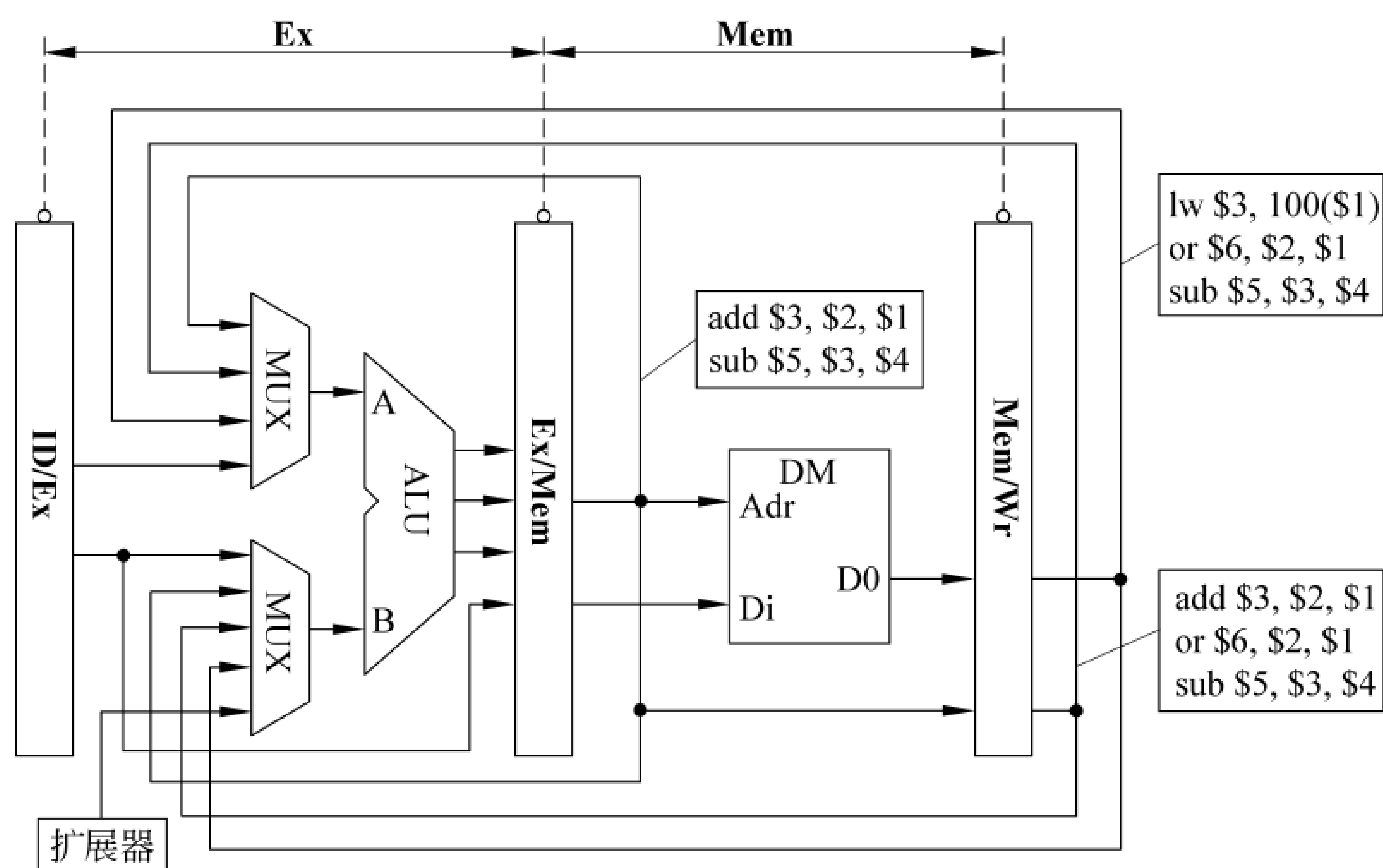


图 7.15　引入转发技术后数据通路中增加的转发线路

从图 7.15 可看出，采用转发技术的数据通路中，在执行阶段 ALU 的两个输入端处，多路选择器的控制信号需要考虑转发条件，因而需要对图 7.7 中的执行部件进行以下部分调整。

(1) 原来 ALU 的 A 输入端加一个三选一多路选择器，随之增加一个两位的控制信号 ALUSrcA。

(2) 原来 ALU 的 B 输入端多路选择器要调整为四选一，原来的一位控制信号 ALUSrc 改为两位控制信号 ALUSrcB。

(3) 控制信号 ALUSrcA 和 ALUSrcB 的取值除了考虑原来的控制信号 ALUSrc 以外，

还要考虑转发条件检测的结果。

(4) 对图 7.15 中的转发线路进行合并，在 Wr 阶段用一个多路选择器将 ALU 输出结果和数据存储器的输出数据合并成一路数据同时转发到 ALU 的两个输入端。

调整后的部分流水线数据通路如图 7.16 所示。

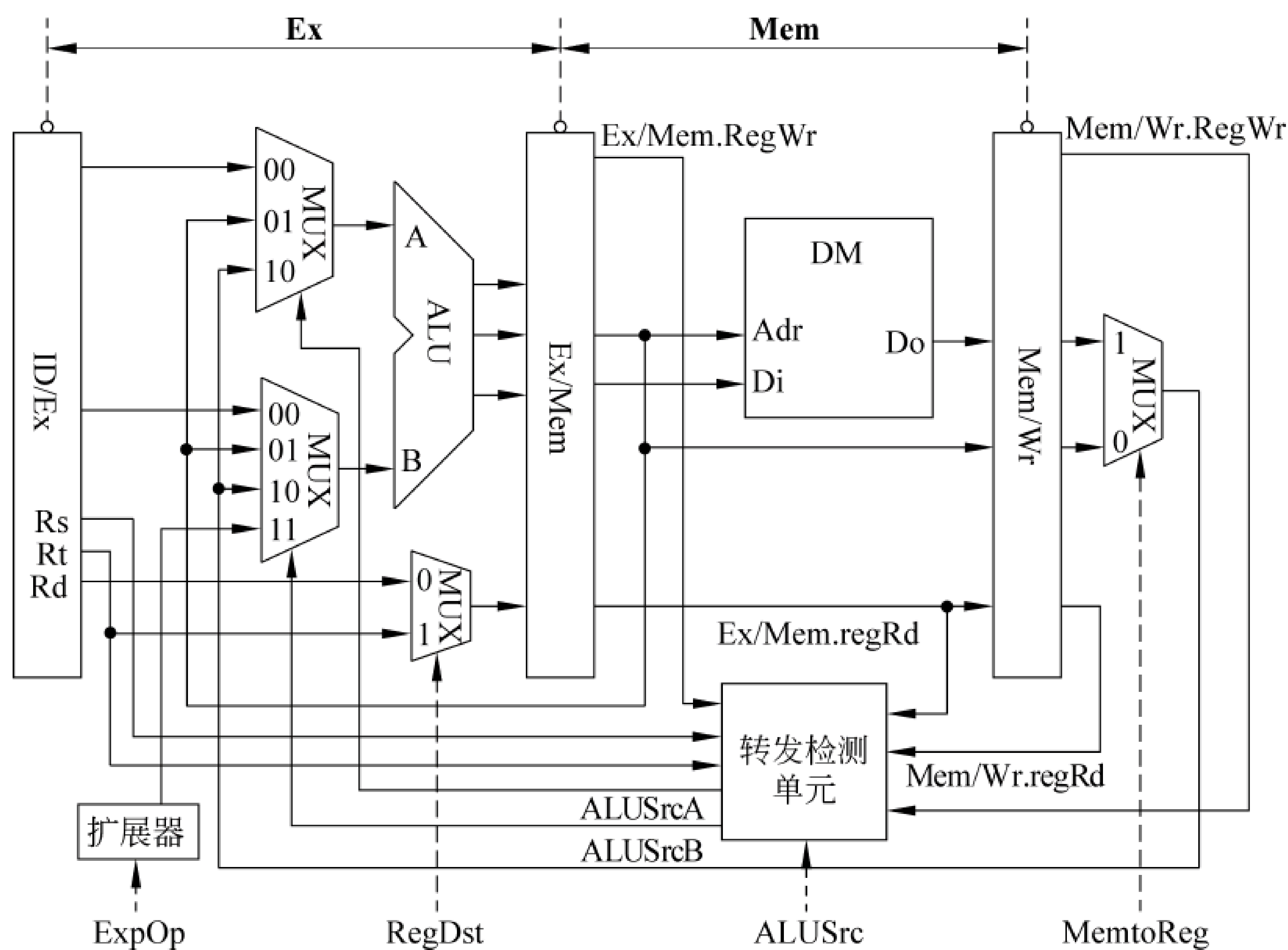


图 7.16　带转发控制的部分流水线数据通路

图 7.16 中转发检测单元的控制逻辑可根据指令的数据相关性来设计。从图 7.15 可以看出，发生数据相关的情况有以下两种。

(1) 本条指令的目的操作数是随后第一条指令所用的源操作数，对应的转发条件如下。

```
C1(A)=(Ex/Mem.regRd=ID/Ex.regRs)
C1(B)=(Ex/Mem.regRd=ID/Ex.regRt)
```

当 C1(A)=1 时，ALUSrcA 应等于 01；当 C1(B)=1 时，ALUSrcB 应等于 01。

(2) 本条指令的目的操作数是随后第二条指令所用的源操作数，对应的转发条件如下。

```
C2(A)=(Mem/Wr.regRd=ID/Ex.regRs)
C2(B)=(Mem/Wr.regRd=ID/Ex.regRt)
```

当 C2(A)=1 时，ALUSrcA 应等于 10；当 C2(B)=1 时，ALUSrcB 应等于 10。

当 C1(A)和 C2(A)都不等于 1 时，ALUSrcA 应等于 00，当 C1(B)和 C2(B)都不等于 1 时，ALUSrcB 应根据 ALUSrc 的值确定，当 ALUSrc=0 时，ALUSrcB 等于 00；当 ALUSrc=1 时，ALUSrcB 等于 11。

以上考虑的仅是基本数据相关情况。实际上，转发条件还要考虑其他一些约束情况，以下是一些例子。

(1) 运算结果不写入目的寄存器。例如，对于 beq 指令后面紧跟一条 ALU 运算类指令的情况，虽然可能满足上述条件，但是 beq 指令并不改变目的寄存器 Rt 的值，所以，不能进

行转发。

(2) 目的寄存器为$0。例如,对于指令 add $0, $7, $8,根据图 7.16 可知,转发的是$7 和$8 的内容相加的结果,可能是一个非 0 数,但实际上下条指令的操作数应该是$0 的内容 0。

(3) 多条连续指令关于同一个寄存器数据相关。例如,对于下列指令序列:

```
add  $1, $1, $2
add  $1, $1, $3
add  $1, $1, $4
...
```

按照前述 C1(A)和 C2(A)的逻辑表达式,可得 C1(A)=C2(A)=1,这样,使得 ALUSrcA 的取值不确定而发生错误。显然,这种情况下,应该使 C1(A)=1,C2(A)=0,也即,当本条指令源操作数和上条指令的目的操作数一样,则不能转发上上条指令的结果,而必须转发上条指令的结果。

综合考虑上述各种情况,得到改进的转发条件逻辑表达式如下。

```
C1(A)=Ex/Mem.RegWr and (Ex/Mem.regRd≠0) and (Ex/Mem.regRd=ID/Ex.regRs)
C1(B)=Ex/Mem.RegWr and (Ex/Mem.regRd≠0) and (Ex/Mem.regRd=ID/Ex.regRt)
C2(A)=Mem/Wr.RegWr and (Mem/Wr.regRd≠0) and
      (Ex/Mem.regRd≠ID/Ex.regRs) and (Mem/Wr.regRd=ID/Ex.regRs)
C2(B)=Mem/Wr.RegWr and (Mem/Wr.regRd≠0) and
      (Ex/Mem.regRd≠ID/Ex.regRt) and (Mem/Wr.regRd=ID/Ex.regRt)
```

通过上述转发条件的检测和对相应的转发线路的控制,可以解决大部分 RAW 数据冒险。

* 4. Load-use 数据冒险的检测和处理

转发能够解决大部分 RAW 数据冒险,那么,lw 指令随后跟 R-型指令或 I-型运算类指令的相关性问题,能否通过转发来解决呢? 如图 7.17 所示,lw 指令只有在 Mem 段结束时才能得到 DM 中的结果,然后送 Mem/Wr 寄存器,在 Wr 段前半周期$1 中才能存入新值,但随后的 sub 指令在 Ex 阶段就要取$1 的值,因此,得到的是旧值,而根据图 7.15 的转发线路,ALU 的输入端要么来自上条指令在 Ex 段生成的、存放在 Ex/Mem 寄存器中的值,要么来自上上条指令的执行结果。由此可知,用转发线路无法解决图中 lw 指令和 sub 指令之间的数据相关问题。通常把这种情况称为“Load-use”数据冒险。

对于“Load-use”数据冒险,最简单的做法是由编译器在 Load 指令之后插入“nop”指令来解决,这样,就无须硬件来处理数据冒险问题。当然,最好的办法是在程序编译时通过调整指令顺序以避免出现“Load-use”现象。

例 7.1 以下是某高级语言源程序中的两条赋值语句。

```
a=b+c;
d=e-f;
```

假定 a、b、c、d、e、f 都被分配在内存,其地址分别用[a]、[b]、[c]、[d]、[e]、[f]表示,通过编译器编译后,生成的汇编目标代码(为说明方便起见,在第一列加了序号)如下。

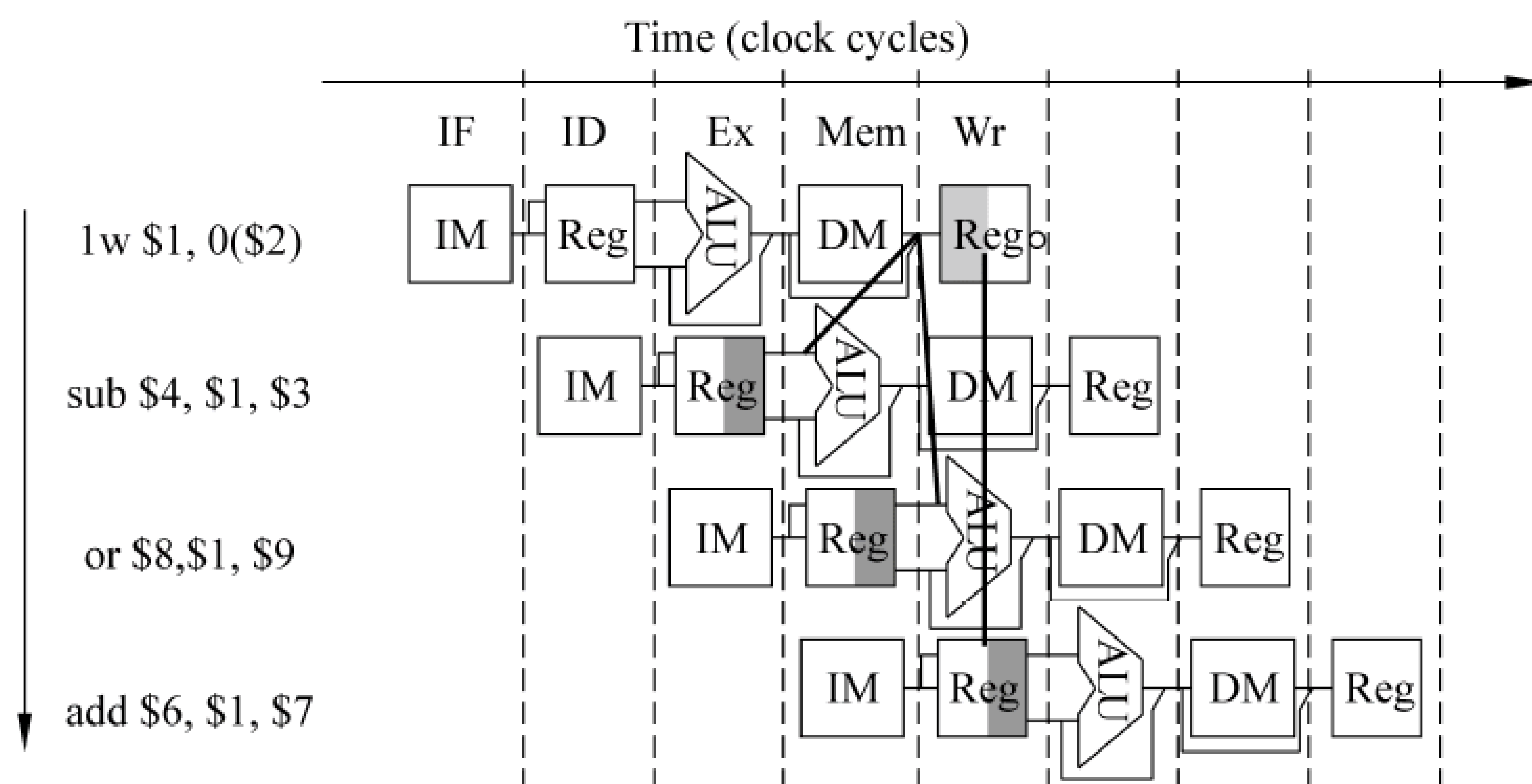


图 7.17　Load-use 数据冒险

```
1.    lw     $2, [b]
2.    lw     $3, [c]
3.    add    $1, $2, $3
4.    sw     $1, [a]
5.    lw     $5, [e]
6.    lw     $6, [f]
7.    sub    $4, $5, $6
8.    sw     $4, [d]
```

请分析上述目标代码中的数据相关性,并说明哪些相关性引起的数据冒险可通过转发技术解决,哪些不能?并要求进行代码优化,以尽量减少 Load-use 数据冒险。

解:上述目标代码中,发生数据相关的指令对是 1-3、2-3、3-4、5-7、6-7、7-8。其中,2-3 和 6-7 两个指令对之间出现了 Load-use 数据冒险,它们不能通过转发技术解决,其他指令对之间的相关性引起的数据冒险都可通过转发技术解决。

可通过调整指令顺序,将一条无关指令插入 Load 和 R-型指令之间来优化代码,以避免 Load-use 现象。本例中通过将第 5 条指令和第 4 条指令分别插入 2-3 和 6-7 指令对中间来优化。以下是编译优化得到的目标代码。

```
1.    lw     $2, [b]
2.    lw     $3, [c]
5.    lw     $5, [e]
3.    add    $1, $2, $3
6.    lw     $6, [f]
4.    sw     $1, [a]
7.    sub    $4, $5, $6
8.    sw     $4, [d]
```

显然,优化后的指令序列比优化前的指令序列在流水线中执行速度快。据统计,优化调度后,Load-use 冒险引起的阻塞现象大约能降低 1/2～1/3。由此可见,编译优化对程序的性能是非常重要的,而了解指令的功能、指令执行流程和流水线结构等对构造良好的编译器又是极其必要的。

如果需要硬件来处理 Load-use 冒险的话，必须在流水线数据通路中增加 Load-use 冒险检测部件，并在检测到发生 Load-use 冒险时进行流水线阻塞处理。从图 7.17 可以看出，Load-use 冒险发生的条件是，上一条是 Load 指令，并且从存储器装入寄存器的数据是当前指令的源操作数。Load-use 冒险的检测越早越好，但是，再早也要在取出指令之后，因此，其检测点可安排在每条指令的译码(ID)阶段。此时，若是 Load-use 冒险，则 Load 指令应处于执行(Ex)阶段。为了能够确定上一条是否是 Load 指令，引入一个新的控制信号 MemRead，Load 指令时该信号取值为 1，否则取值为 0。

根据上述分析，得到 Load-use 冒险检测条件如下。

```
C= ID/Ex.MemRead and ((ID/Ex.regRt= IF/ID.regRs) or (ID/Ex.regRt= IF/ID.regRt))
```

当 C=1 时，说明发生了 Load-used 数据冒险。检测出 Load-use 数据冒险时，Load 指令后面的一条指令(如 7.17 中的 sub 指令)正在 ID 阶段进行译码和取数操作，下个时钟到来时，译码出来的控制信号和寄存器 Rs、Rt 的值将被送到 ID/Ex 流水段寄存器的输入端，同时，Load 后面的第二条指令(如 7.17 中的 or 指令)处在 IF 阶段，正在根据 PC 的值取指令，下个时钟到来时，取出的指令将被送到 IF/ID 流水段寄存器的输入端。为了避免 Load-use 数据冒险，必须使紧随 Load 后的两条指令停顿一个时钟周期后继续执行。可通过将这两条指令的执行结果清除并让它们延迟一个时钟周期实现。具体来说，就是控制实现以下三个操作：①将 ID/Ex 流水段寄存器中的所有控制信号清 0(相当于插入了一个气泡)，而不是送当时译码出来的控制信号；②保持 IF/ID 流水段寄存器的值不变，而不是送当时取出的指令，这样，使 Load 后面的一条指令继续保存在 IF/ID 流水段寄存器中，在下个时钟周期，该指令重新译码/取数；③保持 PC 的值不变，使 Load 后面的第二条指令在下个时钟周期重新取指令。图 7.18 给出了带转发和 Load-use 冒险处理的部分流水线数据通路。从图中可以看出，当检测到存在 Load-use 数据冒险时，检测部件送出三个控制信号，分别控制上述三个操作的实现。

7.3.3 控制冒险

从图 7.5 给出的流水线数据通路来看，正常情况下，指令在流水线中总是按顺序执行，当遇到改变指令执行顺序的情况时，流水线中指令的正常执行会被阻塞。这种由于发生了指令执行顺序改变而引起的流水线阻塞称为控制冒险(Control Hazards)。各类转移指令(包括调用、返回指令等)的执行，以及异常和中断的出现都会改变指令执行顺序，因而都可能会引发控制冒险。

1. 转移指令引起的控制冒险

如图 7.19 所示是一个由于分支指令(条件转移指令)引起的控制冒险的流水线例子。

图 7.19 中，假定 beq 指令的地址为 12，条件满足时其转移目标地址为 1000。从图 7.5 和图 7.7 可以看出，分支指令 beq 的转移目标地址计算操作在 Ex 段，并在 Mem 段由标志 Zero 和控制信号 Branch 来控制，以确定是否将 PC 的值更新为转移目标地址。因此，在图 7.19 例子中，只有当 beq 指令执行到第 5 时钟结束才能将转移目标地址 1000 送到 PC 的输入端，在第 6 时钟到来后，取出 1000 号单元开始的指令送流水线中执行。此时，紧接在 beq 后面的第 16、20 和 24 三个单元的指令已在流水线中被执行了一部分(图中加斜线的流

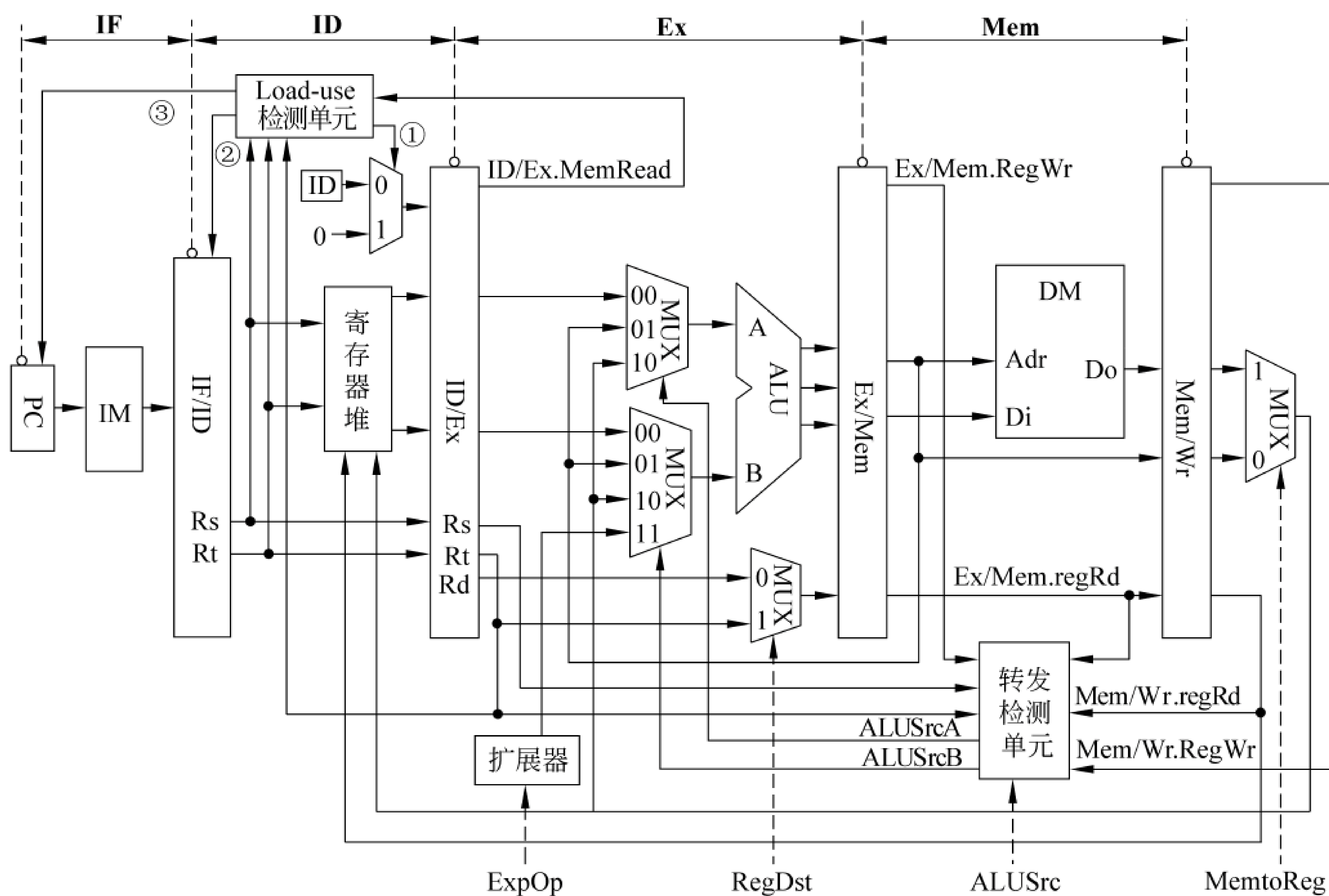


图 7.18　带转发和 Load-use 冒险处理的部分流水线数据通路

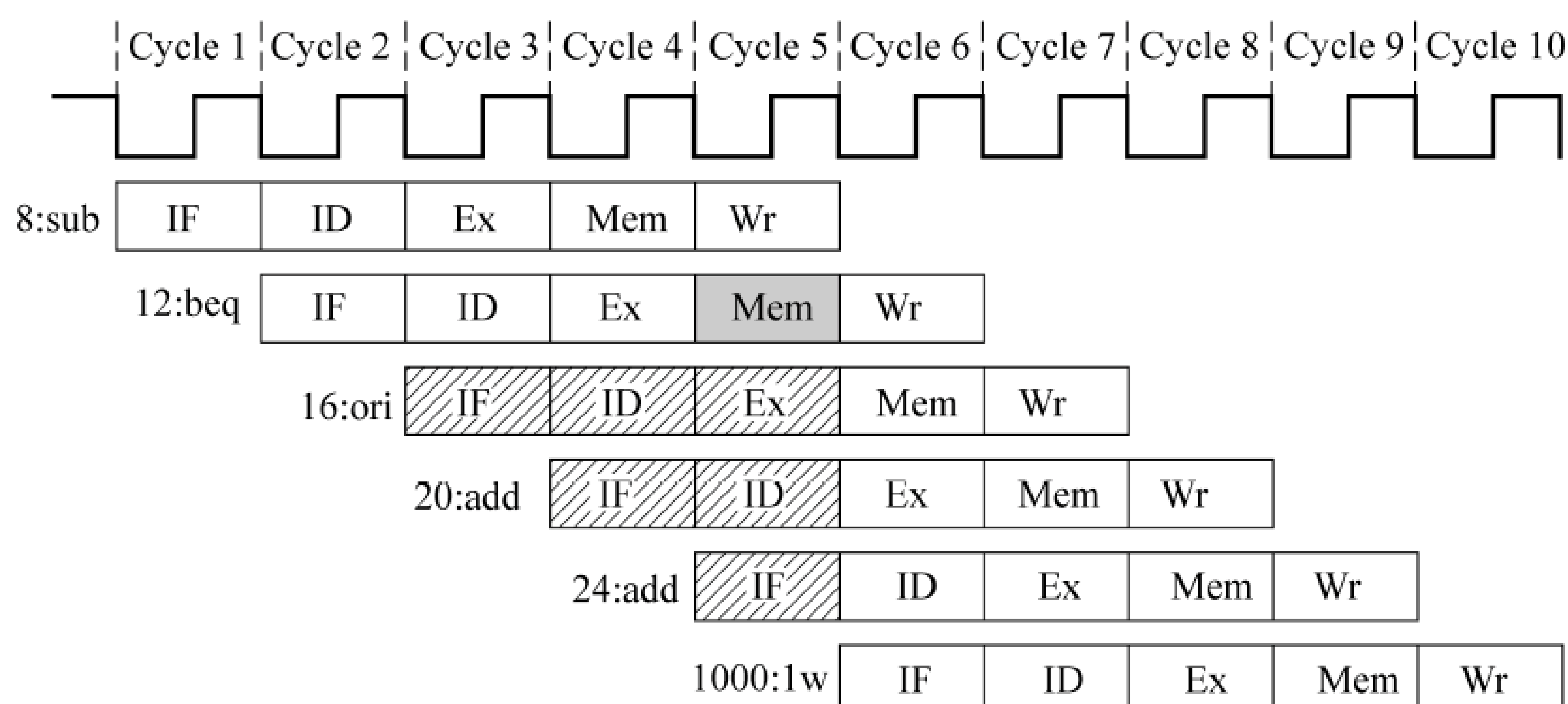


图 7.19　分支指令引起的控制冒险的流水线例子

水段)。显然,正确的执行流程应该是第 12 单元中的 beq 指令执行完后转移到第 1000 单元执行,因此,如果不采取相应措施则指令流水线的执行便发生问题。通常把由于流水线阻塞而带来的延迟执行周期称为延迟损失时间片 C。显然,图 7.19 中的延迟损失时间片 C=3。

由于指令分支而引起的控制冒险也称为分支冒险(Branch Hazards)。对于分支冒险,可采用和前面解决数据冒险一样的硬件阻塞方式(插入气泡)或软件阻塞方式(插入空操作指令)。也即,假设延迟损失时间片为 C,则在数据通路中检测到分支指令时,就在分支指令后插入 C 个气泡,或在编译时在分支指令后填入 C 条 nop 指令。

插入气泡和插入空操作指令这两种都是消极的方式,效率较低。结合分支预测可以降

低由于分支冒险带来的时间损失，分支预测有简单(静态)预测和动态预测两种。此外，还有延迟分支方式也可部分解决分支冒险问题。

*(1) 简单预测

简单预测与指令执行历史无关，因此，它是一种静态预测方式。可以简单预测分支指令的条件总是不满足(not taken)或总是满足(taken)。对于预测不满足的情况，流水线总是按顺序继续执行分支指令的后续指令，如果在数据通路中检测到实际条件确实不满足时，则预测正确，没有任何时间损失；如果检测到实际条件满足时，则预测不正确，此时，将分支指令后续不该执行的指令(如图 7.19 中的第 16、20 和 24 单元中的指令)的控制信号清 0，实际上只需要将寄存器写信号 RegWr 和存储器写信号 MemWr 清 0，就能保证不会改变指令执行结果，相当于执行了空操作。这样，如果分支延迟损失时间片为 3 的话，则预测错误时将损失三个时钟周期。

简单预测方式下，如果转移概率是 50%，则预测正确率仅有 50%。当然，也可以加一些启发式规则来提高简单预测准确率。可以进行一些有条件的简单预测，也即在有些情况下预测总是满足，其他情况预测总是不满足。例如，将循环体顶(底)部的分支总是预测为不满足(满足)。这种方法能达到 65%～85%的预测准确率。

*(2) 动态预测

动态预测(Dynamic Prediction)的准确率可达 90%，现在几乎所有处理器都采用动态预测。它利用分支指令发生转移的历史情况来进行预测，并根据实际执行情况动态调整预测位。转移发生历史情况记录在一个表中，这个表有不同的名称，如分支历史记录表 BHT (Branch History Table)、分支预测缓冲 BPB(Branch Prediction Buffer)、分支目标缓冲 BTB(Branch Target Buffer)等，图 7.20 给出了动态预测和调整过程。

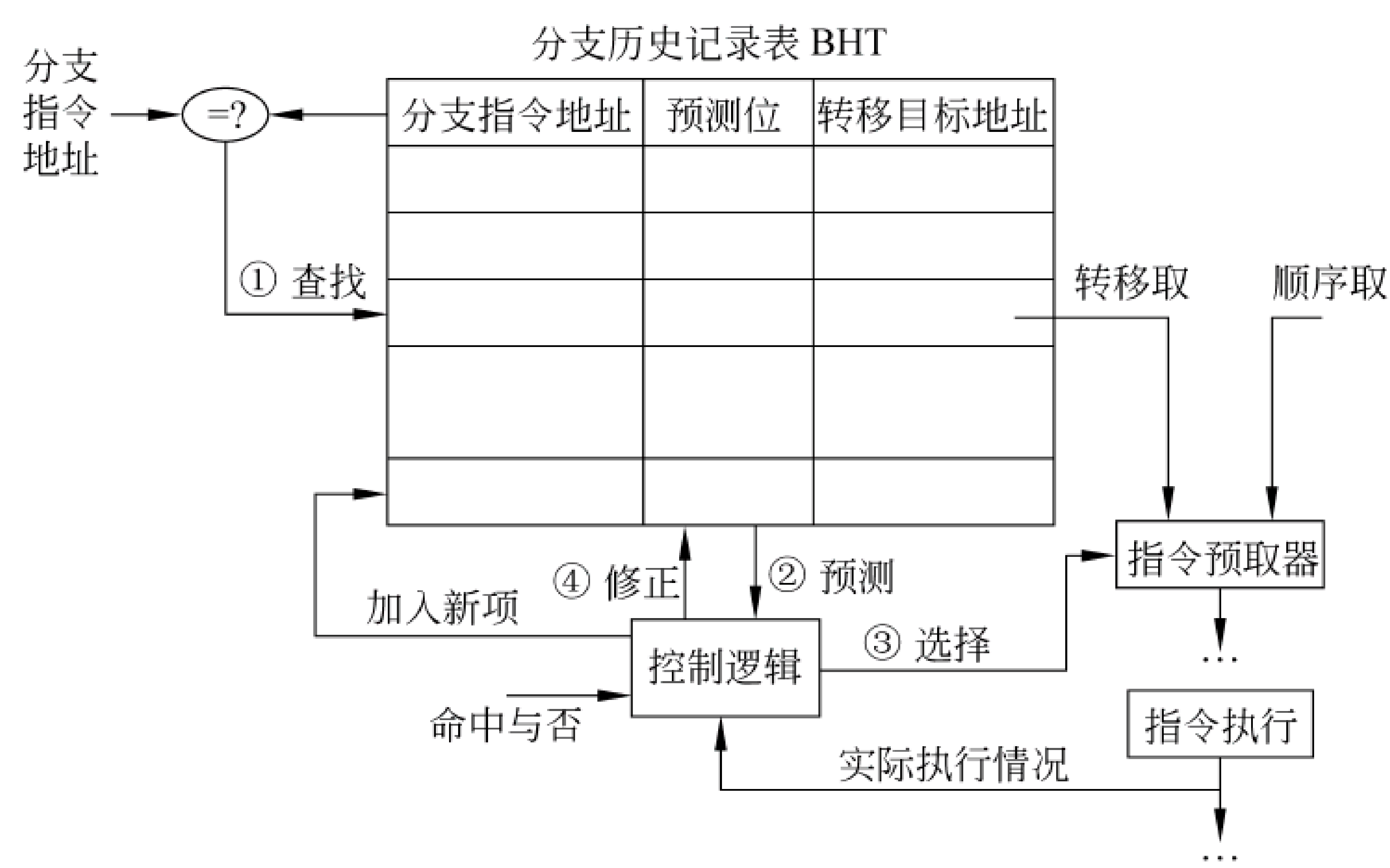


图 7.20　动态预测和调整过程示意图

每个表项由分支指令地址作索引，故在分支指令的 IF 阶段就可取到预测位。因此，完全来得及在分支指令进入 ID 阶段时去取被预测的指令。首先，根据当前分支指令的地址低位查找 BHT 表中对应的项；若未找到(即“未命中”)，说明该分支指令是第一次执行，则由控制逻辑加入一个新项，将该分支指令的地址低位、转移目标地址和初始预测位填入表项

中;若找到(即"命中"),则控制逻辑根据预测位,确定是"转移取"还是"顺序取";在分支指令执行时,控制逻辑根据实际情况来修改调整预测位。预测位的宽度对动态预测准确率有影响。有一位、两位预测位,也有的系统采用两位以上预测位。

① 一位预测位

采用一位预测位时,总是按上次实际发生的情况来预测下次分支情况,可用"1"表示最近一次发生转移(taken),"0"表示未发生转移(not taken)。

预测时,若预测位为1,则预测下次条件满足,会发生转移;若为0,则预测下次条件不满足,不会发生转移。实际执行时,若预测错,则预测位取反;否则,预测位不变。可用一个简单的预测状态图表示预测位的动态调整过程,如图7.21所示。

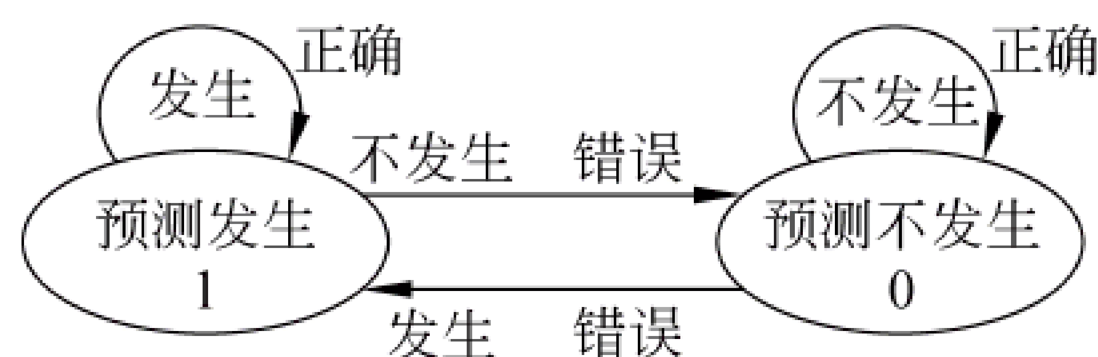


图7.21 一位预测位的状态转换图

采用一位预测位的缺点是,当分支情况连续两次发生改变时,则预测错误。例如,对于循环出口处的分支指令,第一次进循环和最后一次出循环时都会发生预测错误,因为这两次都会改变分支情况;而在循环中每次预测都不会错,因为预测和实际的情况都是发生转移。

例7.2 图7.22给出了某个C语言程序段及其对应的MIPS汇编代码。假定该程序在一个采用一位预测位的流水线处理器上执行,预测位初始为0。试分析当$N=10$和$N=100$时该程序段中各分支指令的预测正确率。

```
int sum (int N)
{
   int i, j, sum=0;
   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         sum=sum+1;
   return sum;
}
```

(a) C语言程序段

```
           …
Loop-i: beq $t1,$a0, exit-i      #若(i=N)则跳出外循环
        add $t2, $zero, $zero    #j= 0
Loop-j: beq $t2, $a0, exit-j     #若(j=N)则跳出内循环
        addi $t2, $t2, 1         #j=j+1
        addi $t0, $t0, 1         #sum=sum+1
        j Loop-j
exit-j: addi $t1, $t1, 1         #i=i+1
        j Loop-i
exit-i: …
```

(b) 汇编程序段

图7.22 循环中分支指令的预测

解:该程序具有两重循环结构,每层循环中有一个分支指令,位于循环入口处。外循环中的分支指令共执行$N+1$次,内循环中的分支指令共执行$N\times(N+1)$次。

预测位初始为0,根据一位预测位状态转换图,可知,外循环中的分支指令只有最后一次预测错误,其余都预测正确;而对于内循环中的分支指令,每次跳出内循环时预测位变为1,再进入内循环时,第一次总是预测错误,并且任何一次循环的最后一次总是预测错误,因此,总共有$1+2\times(N-1)$次预测错误。

当$N=10$时,外循环中分支指令的预测正确率约为$10/11\times100\%=90.9\%$;内循环中分支指令的预测正确率约为$(110-19)/110\times100\%=82.7\%$。

当$N=100$时,外循环中分支指令的预测正确率约为$100/101\times100\%=99\%$;内循环中分支指令的预测正确率约为$(10100-199)/10100\ \times100\%=98\%$。

② 两位预测位

用两位组合成4种情况来表示预测和实际转移的状态，图7.23所示为两位预测位的状态转换图。

4个状态中，有两个状态预测发生转移，有两个状态预测不发生转移。假定11状态表示预测发生(强转移)，实际不发生时，转到状态10(弱转移)，下次仍预测发生转移，如果再次预测错误(即实际不发生)，才使下次预测调整为00状态(强不转移)。从图7.23可看出，只有两次预测错误才改变预测方向。

采用两位预测可避免一位预测时出现的一些问题，使得在连续两次发生不同的分支情况时，也可能会预测正确。

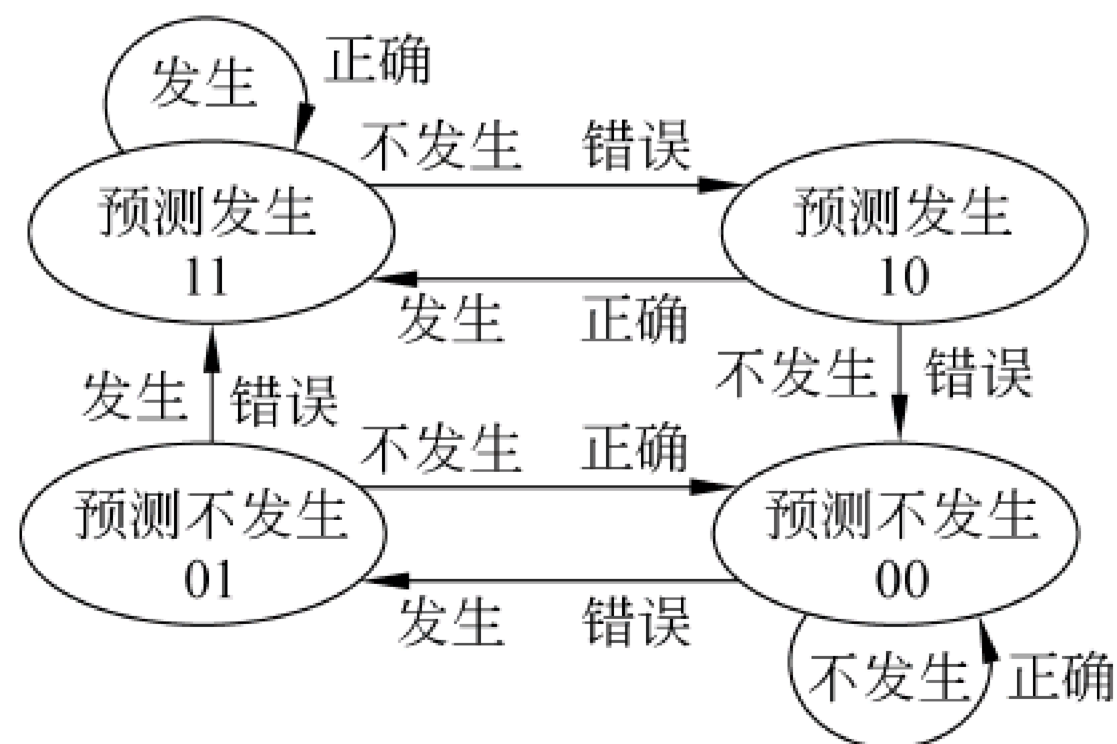


图7.23 两位预测位的状态转换图

例7.3 对于图7.22给出的程序，假定运行在一个采用两位预测位的流水线处理器上，预测位初始为00。试分析当$N=10$和$N=100$时该程序段中各分支指令的预测正确率。

解：预测位初始为00，根据两位预测位状态转换图，可知，外循环中的分支指令只有最后一次预测错误，其余都预测正确；而对于内循环中的分支指令，每次跳出内循环时预测位变为01，预测不发生，再进入内循环时，第一次分支指令实际上也不发生转移，预测正确，而且预测状态变回00，因而又保证下次跳出内循环后再进入时，第一次总是预测正确，所以，对于内循环总是只有最后一次预测错误，总共有N次预测错误。

当$N=10$时，外循环中分支指令的预测正确率约为$10/11\times100\%=90.9\%$；内循环中分支指令的预测正确率约为$(110-10)/110\times100\%=90.9\%$。

当$N=100$时，外循环中分支指令的预测正确率约为$100/101\times100\%=99\%$；内循环中分支指令的预测正确率约为$(10100-100)/10100\times100\%=99\%$。

由此可见，两位预测位方式下，内循环分支指令的预测正确率基本上与外循环的相当。而一位预测位方式下，内循环中分支指令的预测正确率远不及外循环分支指令的预测正确率。

目前，采用比较多的是两位预测位，也有的系统采用两位以上预测位，如Pentium 4的BTB2采用了四位预测位。

注意，采用分支预测方式时，流水线控制逻辑必须确保错误预测指令的执行结果不能生效，而且要能从正确的分支地址处重新启动流水线工作。

*(3) 延迟分支

除了上述介绍的预测结合硬件阻塞和软件插入空指令的方法外，还可以采用延迟分支(Delayed Branch)的方法来解决分支冒险。其主要思想是，采用编译优化来调整指令顺序，把分支指令前与分支指令无关的指令调到分支指令后面执行，以填充延迟损失时间片，不够时用nop操作填充。分支指令后面被填的指令位置称为分支延迟槽(branch delay slot)，需要填入的指令条数(即分支延迟槽数)等于延迟损失时间片。

因为延迟分支技术通过编译器重排指令顺序来实现，所以它属于静态调度技术。图7.24给出了一个分支延迟调度的例子。该例假定流水线的分支延迟损失时间片为2。从图7.24可

看出，在 beq 指令前的所有指令中，可以插到 beq 指令后、add 指令前的只有第一条 lw 指令和 sub 指令，但是，如果把这两条指令都调过去的话，则第二条 lw 指令和 beq 指令就会形成 Load-use 数据冒险，因此，只能有一条指令填入分支延迟槽，另外还需再加一条 nop 指令，以填充分支延迟损失时间片。

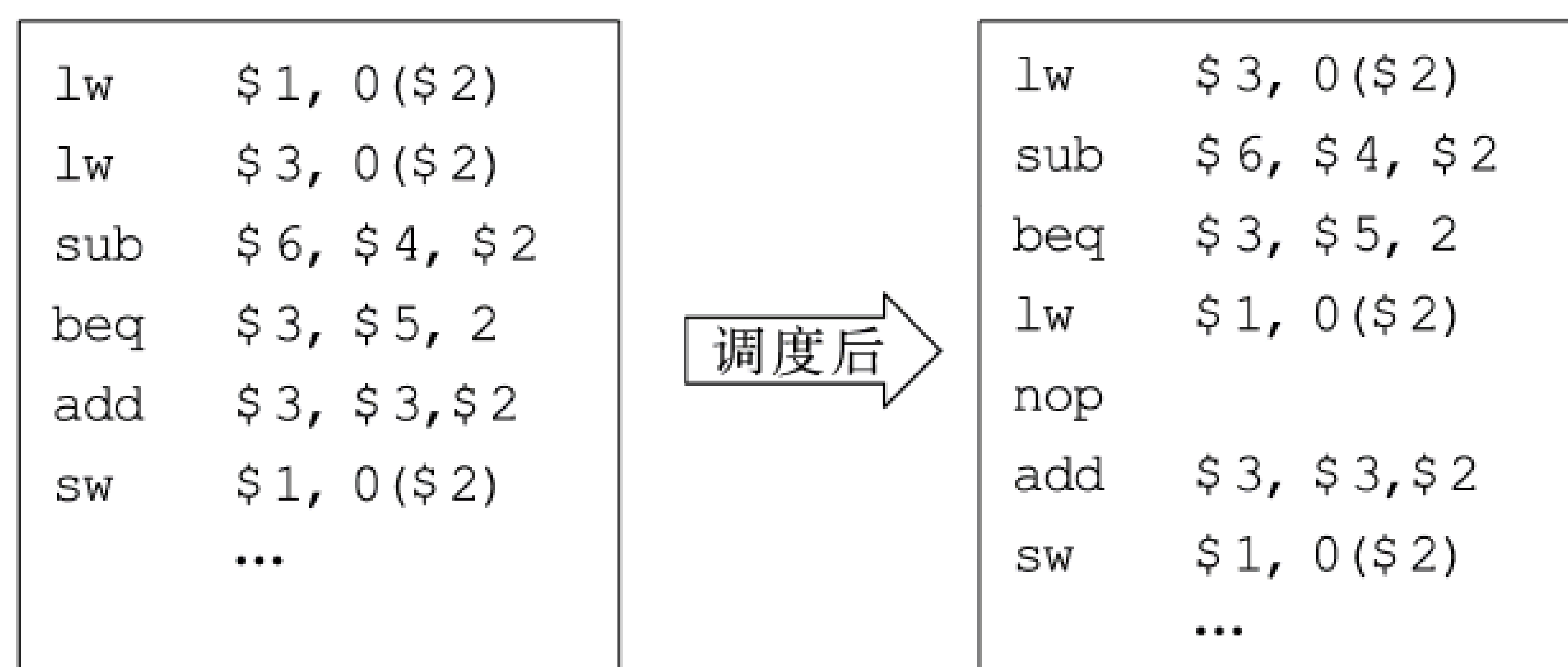


图 7.24　分支延迟调度的一个例子

对于分支冒险来说，分支延迟损失时间片是影响流水线执行效率的重要因素。分支延迟损失时间片越小，插入的气泡或 nop 指令越少，在预测错误时后退的指令条数越少，在分支延迟方式下，调度到分支延迟槽的无关指令条数越少。例如，图 7.24 中假定分支延迟损失时间片减少为 1，则不需填入 nop 指令。

减少分支延迟损失时间片的关键是尽量提早进行分支条件的检测。例如，在前面图 7.5 所示的流水线数据通路中，分支指令 beq 的条件检测在 Mem 阶段进行，因而得到分支延迟损失时间片为 3。如果把检测操作往前调到 Ex 段，甚至提前到 ID 段，则可将分支延迟损失时间片减少到 2，甚至减为 1。

***2. 异常或中断引起的控制冒险**

除了上述介绍的由于分支指令引起的控制冒险外，还有异常或中断引起的控制冒险。

异常和中断的出现会改变程序的执行流程，使得流水线执行发生阻塞。与分支冒险一样，当某条指令执行过程中发现异常或中断时，可能它后面的多条指令已经被取到流水线中正在执行。例如，ALU 运算类指令发现“溢出”时，已经到 Exec 阶段的结束了，此时，它后面已有两条指令进入了流水线。

通过在数据通路的不同流水段中加入相应的检测逻辑可检测出哪条指令发生了异常。例如，“溢出”可在 Exec 段检出；“无效指令”可在 ID 段检出；“除数为 0”可在 ID 段检出；“无效指令地址”可在 IF 段检出；“无效数据地址”可在 Load/Store 指令的 Exec 段检出。检测出异常的那个流水段正在执行的指令，就是发生异常的指令。外部中断的检测可以放在任意一个流水段中进行。若放在 IF 中检测，因为可在取指令前进行检测，所以，如果发现有中断请求发生，则能确保在该时钟周期就开始执行中断服务程序，并让已经在流水线中的指令继续执行完，不需要进行指令冲刷；若放在 Wr 阶段进行，则需要将刚执行完的指令后面几条指令从流水线中清除掉。

对于图 7.5 所示的 5 段流水线处理器，任何一个时钟周期都有 5 条活动的指令，因而很可能在一个时钟周期内同时有多条指令发生异常或中断，不同流水段发生不同类型的异常。例如，在 Ex 阶段 add 指令发生“溢出”的同时，ID 阶段的指令发生了“无效指令”，Mem 阶段的 lw 指令发生了“缺页”，并且又发生了外设 I/O 中断请求。上述这种情况下，显然应该先

响应和处理 lw 指令的“缺页”异常。对于这种同时发生多个异常和中断的情况，最关键的问题是要确定哪条指令的异常应最先被响应和处理。显然，排在前面的指令发生的异常的响应优先级高，因此，优先级确定原则是，在同一个时钟周期内的指令序列中，最先执行的指令所产生的异常最先被响应，外部中断请求最后响应。也即对同时在 5 个指令流水段中发生的异常进行排序时，其顺序为 Wr >Mem>Ex>ID>IF。

处理器硬件对异常和中断引起的冒险的处理，大致的做法如下：当检测到有异常或中断后，首先，清除发生异常的指令以及其后在流水线中的所有指令，然后保存断点，并将异常处理程序的首地址送 PC 的输入端。指令清除的方式和上述分支预测错误时指令清除方式类似，通常是通过相应的冲刷(Flush)控制信号将指令或指令的控制信号清 0(主要保证写信号 RegWr 和 MemWr 清 0)来实现。

许多处理器都能提供一种精确的异常和中断的方式。所谓精确的异常和中断，是指处理器能够确定异常和中断发生的精确位置，也即处理器响应异常和中断时，所保存的断点是精确的返回地址。因为以流水线方式执行指令时，异常可能发生在不同的阶段，因而会产生一些潜在的危险。例如，在图 7.5 所示的 5 段流水线处理器中，假定正在执行指令序列“lw-add-ori-…”，而且 lw 指令将在 Mem 段发生“缺页”，add 指令将在 Ex 段发生“溢出”，ori 指令将在 IF 段发生“指令地址越界”。这种情况下，按正确的处理顺序，应该先处理 lw 指令的异常。但是，因为 ori 指令处于 IF 段时，lw 才处于 Ex 段，add 才处于 ID 段，因此，此时 ori 前面的两条指令都还没有发生异常。如果马上就处理 ori 指令的异常，则 add 指令和 lw 指令的异常就被忽略，而导致程序被错误执行。因此，通常的做法是，每个时钟周期内，在多个流水段发生的异常的原因和断点只是被记录到特定的寄存器中，并将发生异常的标记同时记录到流水段寄存器，发生异常的指令继续在流水线中执行，直到执行到最后一个阶段，由最后阶段内的硬件检测本指令是否发生过异常或此时是否有外部中断发生，若有，则清除流水线中后面所有阶段正在执行的指令，然后转到相应的异常处理程序执行。

及时检测到“异常”并处理是非常重要的，否则，会发生错误。例如：在执行 lw \$1，0(\$1)时，若没有及时捕获到“缺页”或没有及时冲刷 RegWr 等控制信号，则可能会使 \$1 改变其值，再重新执行该指令时，所读的内存单元地址可能被改变而发生严重错误。

所有异常处理中，最重要的也是最经常发生的是存储器访问异常，包括“缺页”和“TLB 缺失”等。

*7.3.4 访问缺失引起的流水线阻塞

在使用 cache 的系统中，数据通路中的指令存储器 IM 和数据存储器 DM 分别是 code cache 和 data cache。在流水线处理器中，CPU 执行指令进行取指令或取数据时，如果发生 cache 缺失，则无法立即在 cache 中取到信息而需到主存去取，因而指令执行被阻塞。

前面在介绍 cache 时说过，cache 中有相应的“命中”检测线路(用主存地址高位与 cache 标志比较)。当不命中时，进入以下流水线阻塞处理过程：冻结所有临时寄存器和程序员可见寄存器的内容，由一个单独的控制器处理 cache 缺失。

若是指令缺失，则把发生缺失的指令所在的主存块的首地址送地址总线启动一次“主存块读”总线事务，并等待主存完成一个主存块的读操作，把读出的一个主存块写到 cache 对应行的数据区；若对应 cache 行全满的话，还要考虑淘汰掉一个已在 cache 中的主存块；把地

址高位部分(标记)写到 cache 的“tag”字段,并置“有效位”;最后,重新执行该指令的第一步,即从“取指令”阶段开始。

若是读数据缺失,其处理过程和指令缺失类似,但从主存读出数据后,从“取数”那一步开始重新执行就可以了;若是写数据缺失,则要考虑用哪种“写策略”解决“一致性”问题。

cache 缺失引起的流水线阻塞比数据相关或分支指令引起的流水线阻塞简单,只要保持所有寄存器不变即可。与异常引起的阻塞不同,它不需要进行程序切换,因而无须保存断点。

同样,TLB 缺失和“缺页”也会引起流水线阻塞。若 TLB 缺失由硬件处理的话,则处理过程类似于 cache 缺失处理。但有的系统将 TLB 缺失当成一种异常,由软件来处理,此时,TLB 缺失和“缺页”一样,采用上一节提到的异常冒险处理方式进行。

至此我们已经讨论了单周期、多周期和流水线三种处理器实现方式,指令在这三种处理器上采用不同的执行方式,得到不同的执行效率。从下面的例子中可以看出,虽然单周期和理想流水线两种方式下的 CPI 都是 1,但时钟周期的宽度相差很大,因而,同样的程序所用时间相差很大。

例 7.4 假设数据通路中各主要功能单元的操作时间如下。存储器:400ps;ALU 和加法器:200ps;寄存器堆读口或写口:100ps。假设 MUX、控制单元、PC、扩展器和传输线路等的延迟忽略不计,程序中指令的组成比例为:取数指令占 25%;存数指令占 10%;ALU 运算类指令占 52%;分支指令占 11%;跳转指令占 2%。则下面的实现方式中,哪个更快?快多少?

(1) 单周期方式(如图 6.24 所示),每条指令在一个固定长度的时钟周期内完成。

(2) 多周期方式(如图 6.33 所示),并且每类指令时钟数为:取数—5,存数—4,ALU—4,分支—3,跳转—3。

(3) 流水线方式(如图 7.5 所示),具体来说,每条指令分取指令、取数/译码、执行、存储器访问和写回 5 个阶段。假定没有结构冒险;数据冒险采用“转发”技术处理;分支延迟损失时间片为 1,预测准确率为 75%;不考虑异常、中断和访问失效引起的流水线阻塞。

解:CPU 执行时间=指令条数×CPI×时钟周期,对于同一个程序,三种方式的指令条数都一样,因此只要比较 CPI 和时钟周期即可。

根据已知条件,得到各类指令实际需要的执行时间如下。

取数指令:取指 400ps+寄存器读 100ps+ALU 运算 200ps+取数 400ps+寄存器写 100ps=1.2ns。

存数指令:取指 400ps+寄存器读 100ps+ALU 运算 200ps+存数 400ps=1.1ns。

ALU 指令:取指 400ps+寄存器读 100ps+ALU 运算 200ps+寄存器写 100ps=800ps。

分支指令:取指 400ps+寄存器读 100ps+ALU 运算 200ps=700ps。

跳转指令:取指 400ps 。

(1) 单周期方式下,时钟周期由最长的取数指令确定,为 1.2ns。因此,N 条指令的执行时间为 1.2N ns。

(2) 多周期方式下,以功能部件最长所需时间作为时钟周期,因为存储器访问操作时间最长,为 400ps,所以,时钟周期为 400ps。根据各类指令的频度,计算出平均时钟周期数如下。

$$5\times25\%+4\times10\%+4\times52\%+3\times11\%+3\times2\%=4.12$$

所以，N 条指令的执行时间为 $4.12\times400\text{ps}\times N=1.648N\text{ns}$。

(3) 流水线方式下，流水线的时钟周期取功能部件最长所需时间为 400ps。每类指令所需的时钟数如下。

取数指令：当发生 Load-use 冒险时，执行时间为 2 个时钟周期，否则为 1 个时钟周期，故平均执行时间为 1.5 个时钟周期。

存数指令、ALU 指令：因为采用了"转发"机制，所以流水线不会被阻塞，故只需 1 个时钟周期。

分支指令：分支延迟损失时间片为 1，因而预测错误时阻塞 1 个时钟周期。这样，预测成功时需 1 个时钟周期，预测错误时需 2 个时钟周期。平均约为 $0.75\times1+0.25\times2=1.25$ 个时钟周期。

跳转指令：需等到译码阶段结束才能得到转移地址，故需 2 个时钟周期。

因此，平均 CPI 为 $1.5\times25\%+1\times10\%+1\times52\%+1.25\times11\%+2\times2\%=1.17$。

所以，N 条指令的执行时间为 $1.17\times400\text{ps}\times N=0.468N\text{ns}$。

综上所述，流水线方式的执行速度最快，与单周期相比，约为 1.2ns/0.468ns=2.56 倍；与多周期相比，约为 1.648ns/0.468ns=3.52 倍，都要快一倍以上。

细心的读者可能发现，第 6 章中例 6.1 对单周期和多周期相比时，结论是多周期比单周期快 1.23 倍，而从本例来看，单周期比多周期更快。得到这种矛盾的结论，是因为假定的前提不同。本例中，访存操作消耗了特别长的时间，它是多周期方式和流水线方式共同的瓶颈。因为各功能段耗时不均匀，所以会导致单周期方式比多周期快的现象。若将访存分解为两个周期，可以使时钟周期降为 200ps，这对多周期和流水线两种方式都会带来性能上的改进。

第 7.4 节介绍的超流水线技术就采用了将流水段更细、更均匀地划分的思想。

7.4 高级流水线技术

高级流水线技术充分利用指令级并行(ILP)来提高流水线的性能。有两种增加指令级并行的策略。

一种是超流水线(super-pipelining)技术，通过增加流水线级数来使更多的指令同时在流水线中重叠执行。超流水线并没有改变 CPI 的值，CPI 还是 1，但是，因为理想情况下流水线的加速比与流水段的数目成正比，因此，流水段越多，时钟周期越短，指令吞吐率越高。因此，超流水线的性能比普通流水线好。但是，流水线级数越多，用于流水段寄存器的开销就越大，因而流水线级数是有限制的，不可能无限增加。

另一种是多发射流水线(multiple issue pipelining)技术，通过同时启动多条指令(如整数运算、浮点运算、存储器访问等)独立运行来提高指令并行性。采用多发射流水线技术的处理器称为超标量(Superscalar)处理器。要实现多发射流水线，其前提是数据通路中有多个执行部件，如定点、浮点、乘除、取数/存数部件等。多发射流水线的 CPI 能达到小于 1，因此，有时用 CPI 的倒数 IPC 来衡量其性能。IPC(Instructions Per Cycle)是指每个时钟周期内完成的指令条数。例如，4 路多发射流水线的理想 IPC 为 4。

实现多发射流水线必须完成以下两个任务：指令打包和冒险处理。指令打包任务就是将能够并行处理的多条指令同时发送到发射槽中，因此处理器必须知道每个周期能发射几条指令，哪些指令可以同时发射。这通过推测(speculation)技术来完成，可以由编译器或处理器通过猜测指令执行结果来调整指令执行顺序，使指令的执行能达到最大可能的并行。指令打包的决策依赖于"推测"的结果，主要根据指令间的相关性来进行推测，与前面指令不相关的指令可以提前执行，例如，如果可以推测出一条 Load 指令和它之前的 Store 指令引用的不是同一个存储地址，则可以将 Load 指令提前到 Store 指令之前执行；也可对分支指令进行推测以提前执行分支目标处的指令。不过，推测仅是"猜测"，有可能推测错误，故需有推测错误检测和回退机制，在检测到推测错误时，能回退掉被错误执行的指令。因此，错误推测会导致额外开销。需要结合软件推测和硬件推测来进行，软件推测指编译器通过推测来静态重排指令，此种推测一定要正确，而硬件推测指处理器在程序执行过程中通过推测来动态调度指令。

根据推测打包任务主要由编译器静态完成还是由处理器动态执行，可将多发射技术分为两类：静态多发射和动态多发射。

*7.4.1 静态多发射处理器

静态多发射处理器主要通过编译器静态推测来辅助完成"指令打包"和"冒险处理"。指令打包的结果可看成将同时发射的多条指令合并到一个长指令中。通常将一个周期内发射的多个指令看成一条多个操作的长指令，称为一个"发射包"。所以，静态多发射指令最初被称为"超长指令字"(Very Long Instruction Word，VLIW)，采用这种技术的处理器被称为 VLIW 处理器。Intel 的 IA-64 架构采用这种方法，Intel 称其为 EPIC(Explicitly Parallel Instruction Computer，显式并行指令计算机)。

因为数据通路中功能部件及其个数是确定的，所以，同一时钟周期内发射的指令类型和指令个数是受限的。例如，如果 ALU 运算部件和存储访问部件独立设置，并只各提供一套，则只能同时发射一条 ALU 指令/分支指令和一条 Load/Store 指令。

静态多发射处理器的冒险处理主要是数据冒险和控制冒险。处理冒险的方式可有两种。一种是完全由编译器通过代码调度和插入 nop 指令来静态地消除所有冒险，无须硬件实现冒险检测和流水线阻塞；另一种是由编译器通过静态分支预测和代码调度来消除打包指令的内部依赖，而由硬件检测数据冒险并进行流水线阻塞。

为了使读者对静态多发射处理器有一定的感性认识，下面以一个简单的 2-发射 MIPS 处理器为例来分析静态多发射处理器的基本实现，然后简要介绍一下 Intel 的 IA-64 的结构特点。

1. 2-发射 MIPS 处理器

要使原来的 MIPS 处理器能够同时处理两条流水线，数据通路必须进行一些改进。

(1) 因为需要同时读取并译码两条指令，因而，可将两条指令打包成 64 位长指令，前面为 ALU/beq 指令，后面为 Load/Store 指令，没有配对指令时，就用 nop 指令代替，将 64 位长指令中的两个操作码同时送到控制器(指令译码器)进行译码。

(2) 因为两条指令可能同时读两个寄存器(与 Store 配对时)或同时写两个寄存器(与 Load 配对时)，所以需要增加一个寄存器读口和一个写口。

(3) 因为在上一条 ALU/beq 指令进行 ALU 运算时，本条 Load/Store 指令要计算地址，所以需要增加一个加法器或 ALU 运算部件(包括两组输入总线和一组输出总线)。

(4) 流水段寄存器要增宽，因为两条指令的数据信息和控制信号在流水段寄存器中被分别传送。

2-发射 MIPS 处理器的潜在性能将提高大约两倍，但由于各种原因，实际上达不到。静态多发射处理器的缺点是，为消除结构冒险，需增加额外部件，此外，由于多条流水线同时发射执行，使得一旦发生数据冒险或控制冒险便会有更多的指令被阻塞在流水线中，因而增加了潜在的性能损失。例如，对于 Load-use 数据冒险，在单发射流水线下，只有一条指令被延迟执行，而在 2-发射流水线下，因为一个时钟周期有两条指令在执行，所以，有两条指令被延迟执行；又例如，对于 ALU-Load/Store 数据冒险(即一条 ALU 指令后跟一条 Load 或 Store 指令)，在单发射流水线下，可用“转发”技术使 ALU 结果直接转发到 Load/Store 指令的 Ex 阶段，但在 2-发射流水线下，因为两条指令同时进行，所以，ALU 的结果不能直接转发，只能延迟一个时钟周期再执行 Load/Store 指令。为了更有效地利用多发射处理器的并行性，必须有更强大的编译器，能够充分消除指令间的依赖关系，使指令序列达到最大的并行性。

以下用一个例子来说明编译器如何进行静态指令调度。假设有一个程序段用来实现对一个数组中的所有元素依次进行加 1 操作，该程序段在 MIPS 机器上对应的机器代码段如下。

```
loop: lw      $t0, 0($s1)          #从存储单元取数,送$t0
      addiu   $t0, $t0, 1          #$t0增量
      sw      $t0, 0($s1)          #$t0送回存储单元
      addi    $s1, $s1, -4         #存储单元地址减 4
      bne     $s1, $zero, loop     #若存储单元地址不为 0,则继续循环
```

由于受 2-发射 MIPS 流水线处理器结构的限制，指令被分成两类。一类是 ALU/分支指令；一类是 lw/sw 指令。为了能在 2-发射 MIPS 流水线中有效执行上述程序，需重新排列指令序列。前三条指令之间和后两条指令之间各有相关性，因此，可把第四条指令调到第一条后面，但因为 $s1 先被减 4，故 sw 指令的偏移应改为 4。调度后得到的 2-发射流水线如图 7.25 所示。

	ALU/分支指令	Load/Store 指令	时钟周期
loop:	nop	lw　$t0, 0($s1)	1
	addi　$s1, $s1, −4	nop	2
	addiu　$t0, $t0, 1	nop	3
	bne　$s1, $zero, loop	sw　$t0, 4($s1)	4

图 7.25　2-发射 MIPS 流水线的指令代码调度例子

上述调度结果是循环体内 5 条指令在 4 个时钟周期内完成，因此，实际 CPI 为 0.8，即 IPC=1.25。这个结果与理想情况相比，相差很大。对于在循环结构中的代码，更好的调度技术是“循环展开”。

循环展开的基本思想是，将循环体展开生成多个副本，在展开的指令中统筹调度。例如，对于上例，若循环执行次数是 4 的倍数，则可循环展开 4 次，其最佳调度序列如图 7.26 所示。

	ALU/分支指令	Load/Store 指令	时钟周期
loop:	addi　$ s1, $ s1, −16	lw　$ t0, 0($ s1)	1
	nop	lw　$ t1, 12($ s1)	2
	addiu　$ t0, $ t0, 1	lw　$ t2, 8($ s1)	3
	addiu　$ t1, $ t1, 1	lw　$ t3, 4($ s1)	4
	addiu　$ t2, $ t1, 1	sw　$ t0, 16($ s1)	5
	addiu　$ t3, $ t3, 1	sw　$ t1, 12($ s1)	6
	nop	sw　$ t2, 8($ s1)	7
	bne　$ s1, $ zero, loop	sw　$ t3, 4($ s1)	8

图 7.26　2-发射 MIPS 流水线的指令代码“循环展开”调度例子

循环展开 4 次后，每次循环体内对 4 个数组元素进行操作，数组首地址在 $ s1 中。因为 addi 指令先将 $ s1 减 16。因此，如图 7.27 所示，循环内被操作的 4 个数组元素的地址分别是($ s1)+16、($ s1)+12、($ s1)+8、($ s1)+4，因为第一个时钟周期的 lw 指令进行地址计算时，addi 指令的执行结果还没写到 $ s1 中，所以，此时 $ s1 还是原来的值，因此，该 lw 指令的地址偏移是 0 而不是 16。

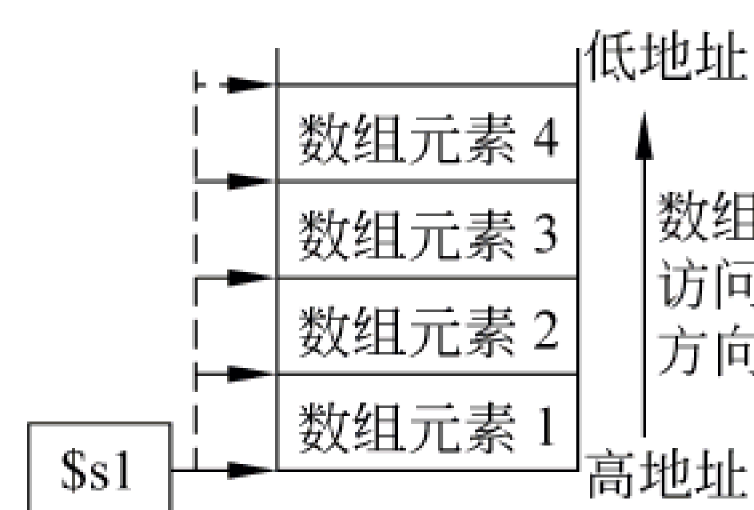


图 7.27　循环展开后的数组元素

循环展开 4 次后，循环体内与数组元素的访问和操作相关的指令(lw, addiu, sw)各有 4 条，再加上 1 条 addi 和 1 条 bne，共 14 条指令，用了 8 个时钟，CPI 达到 8/14=0.57，比未进行循环展开好。

在循环展开过程中，用到了“重命名寄存器”技术，多用了三个临时寄存器 $ t1, $ t2, $ t3 来消除名字依赖关系。因为名字依赖是一种非真实依赖，只是寄存器名相同而已，实际上并不是同一个寄存器，因此，可以用另一个寄存器名替换。

在循环展开时，需要注意尽量不能引起新的数据冒险，例如，第一条 addiu 指令不能放在第二时钟周期，否则，会引起 Load-use 数据冒险。

从上述例子可以看出，循环展开确实能提高程序执行效率，但是，这也是有代价的。本例的代价是多用了三个临时寄存器，并增加了程序的代码长度，因为循环体变长使程序所占的存储空间变大了。

当然，如果上述例子中循环次数不是 4 的倍数，那么就会有问题。这种情况下，除了调整循环次数外，还要对多出来的不足 4 次的循环另外进行处理。

2. Intel 的 IA-64 的结构特点

近 20 年来，Intel IA-32 体系结构一直是市场上最流行的通用处理器架构，1985 年推出的 Intel 80386 处理器是 IA-32 家族中第一款产品，它是典型的 CISC 风格指令集体系结构，而随后的 IA-32 处理器都与它保持向后兼容，因此，从整体上来看，IA-32 仍是基于 CISC 架构的体系结构。但从 1993 年推出的奔腾(Pentium)处理器开始，RISC 设计思想逐渐被引入，Intel 最终将 Pentium 4 处理器设计成了被称为“CISC 壳、RISC 核”架构的处理器，CISC 指令在执行时被转换成一条或多条类似 RISC 指令的微操作。有关 Pentium 4 的具体内容

见 7.4.3 节。

随着计算机技术及应用领域的不断发展,32 位处理器逐步开始向 64 位处理器过渡。Intel 在 IA-32 基础上推出了 EM64T(Extended Memory 64 Technology, 64 位内存扩展技术),它提供了 64 位线性地址空间,将原来的通用寄存器全部扩展为 64 位,并增加了 8 个 64 位通用寄存器,但它不是真正的 64 位结构,而是在 32 位与 64 位之间的过渡模式。真正 64 位架构的是基于 EPIC 技术的 IA-64 体系结构。为了表示 EM64T 的 64 位模式特点,又使两种 64 位结构有所区别,Intel 把 EM64T 改名为 Intel 64。因此,IA-64 和 Intel 64 是两种完全不同的体系结构。

Intel 的安腾(Itanium)和安腾 2(Itanium 2)处理器分别在 2000 年和 2002 年问世,它们是 IA-64 体系结构的最早的具体实现。安腾体系结构完全脱离了 IA-32 CISC 架构的束缚,采用全新的“显式并行指令计算(EPIC)”技术,以最大限度地提高软件和硬件之间的协同性,力求将处理器的处理能力和编译软件的功能结合起来,在指令中将并行执行信息以明显的方式告诉硬件。

它采用类似 64 位 MIPS 架构的 RR 型 RISC 风格指令集,但与 MIPS-64 体系结构有一些差别,它主要有以下几个方面的特点。

(1) 具有比 MIPS-64 更多的寄存器,包含 128 个整数、128 个浮点数、8 个专用分支、64 个一位谓词寄存器。支持寄存器窗口重叠技术,提供一组窗口寄存器,在执行过程调用和返回时,利用窗口寄存器来完成参数传递,即调用程序的输出返回参数与被调用程序输入的入口参数重叠使用同一套寄存器,使得不再需要保存和恢复寄存器内容,可大大提高程序执行的速度。

(2) 要求编译器显式地给出指令级的并行性,并行执行信息被明显标记在代码中。这也是 Intel 之所以称其为显式并行的原因。IA-64 的编译器通过“指令组(instruction group)”来显式地标出指令的并行性。指令组是指相互间没有寄存器级数据依赖的指令序列,指令组长度任意,用“停止标记”在指令组之间明显标识,指令组内部的所有指令可并行执行,只要有足够硬件且无内存操作依赖。

(3) 程序执行时将同时发射的指令重新编码并组织在一个“指令包(bundle)”中,每个指令包的长度为 128 位,由 5 位长的模板字段和三个 41 位长的指令组成。模板字段对应于以下 4 类功能部件中的三条指令:整数 ALU(包括移位和多媒体处理)、访存、浮点和转移分支。由编译器通过 5 位模板字段向硬件指明执行三条指令分别使用哪三个功能部件,同时,编译器还通过模板字段中隐含的“停止标记”来明确地向硬件指明哪条指令是一个指令组的结束位置。41 位指令字中高 14 位为操作码 op,中间三个 7 位寄存器编号分别指出两个源操作数和一个目的操作数所在寄存器号,最后 6 位是一个谓词寄存器的编号。

(4) 引入特殊的谓词化技术,以支持推测执行和消除分支,提高指令级并行度。“谓词”是指分支指令中的条件,每个谓词与一个谓词寄存器相关联。指令最后 6 位是一个标识谓词寄存器的编号,因此,每条指令用该编号与一个谓词寄存器关联,以反映条件是否满足。这样就将指令执行与否与谓词相关联,而不是与分支指令条件关联。谓词化技术可消除循环内的 if-then-else 分支,例如,if (p) {Statement1} else {Statement2}可被编译成以下两条指令。

```
(p) Statement1
```

```
(~p) Statement2
```

上述指令中的 p 和～p 分别存放在两个不同的谓词寄存器中，其指令的含义是，当括号中指明的谓词寄存器的内容为 1 时，执行后面的代码，否则，转化为 nop 指令。由此可见，这两条指令可以完全独立执行，相互没有依赖，而且也无需像传统的分支指令那样在条件满足时跳转到一个目标地址去执行。

以上对静态多发射流水线技术作了简要介绍，可以看出，计算机的体系结构和编译器的关系非常紧密，编译器的好坏直接影响程序的性能。实现编译器的程序员必须对机器结构非常了解，才能开发出质量优良的编译器。

*7.4.2 动态多发射处理器

动态多发射流水线处理器在指令执行时由处理器硬件动态进行流水线调度来完成“指令打包”和“冒险处理”，能在一个时钟周期内执行一条以上指令。

在动态多发射处理器上要达到较好的性能，也需要由编译器进行静态调度，以尽量消除依赖关系，使之达到较高的发射速率。但这种静态调度和静态多发射处理器和 VLIW 处理器的静态调度有一些不同。对于 VLIW 处理器的静态调度来说，编译结果与机器结构密切相关；而对于动态调度来说，编译器仅进行指令顺序调整，不需要根据机器结构进行指令打包，而是完全由硬件来决定某个时钟周期发射哪几条指令。

目前超标量处理器大多采用动态多发射流水线，在简单的超标量处理器中，指令按顺序发射，每个周期由处理器决定是发射一条或多条指令，显然，在这种处理器上要达到较好的性能，很大程度上依赖于编译器。为了更好地发挥超标量处理器的性能，多数超标量处理器都结合动态流水线调度(Dynamic pipeline scheduling)技术。处理器通过指令相关性检测和动态分支预测等手段，投机性地不按指令顺序执行，当发生流水线阻塞时，根据指令的依赖关系，动态地到后面找一些没有依赖关系的指令提前执行。这种指令执行方式称为乱序执行(Out-of-order Scheduling)。

例如，对于以下一段 MIPS 指令序列：

```
lw      $t0, 0($s1)
add     $t2, $t0, $t1
sub     $s2, $s2, $t3
addiu   $t4, $s2, 20
```

第一条和第二条指令存在 Load-use 数据冒险，而且，lw 指令本身耗时较长，也容易发生访问不命中引起的流水线阻塞，因此，可以采用动态流水线调度方式，将 sub 指令调到第二条指令前面提前执行，不需等 lw 和 add 指令执行完。addiu 指令也可以提前，但因为和 sub 指令有依赖关系，所以要保证它在 sub 指令后执行。

为了实现动态多发射和动态流水线调度，处理器中需要提供一些必要的机制和相应的处理部件，如，指令预取部件、指令调度与分派部件、多个功能部件、重排序缓存等。图 7.28 是动态多发射流水线处理器的通用模型示意图。

如图 7.28 所示，动态多发射流水线处理器主要由以下几个部分组成。

(1) 指令预取和译码单元。为了保证流水线中有足够的指令执行，必须要有指令预取

功能。预取的指令经译码后,放到一个指令队列中。

(2) 指令分派(dispatch)器。通过分析指令功能和指令间的依赖关系,并根据功能部件的空闲情况,确定何时、发射哪条指令、到哪个功能单元中。

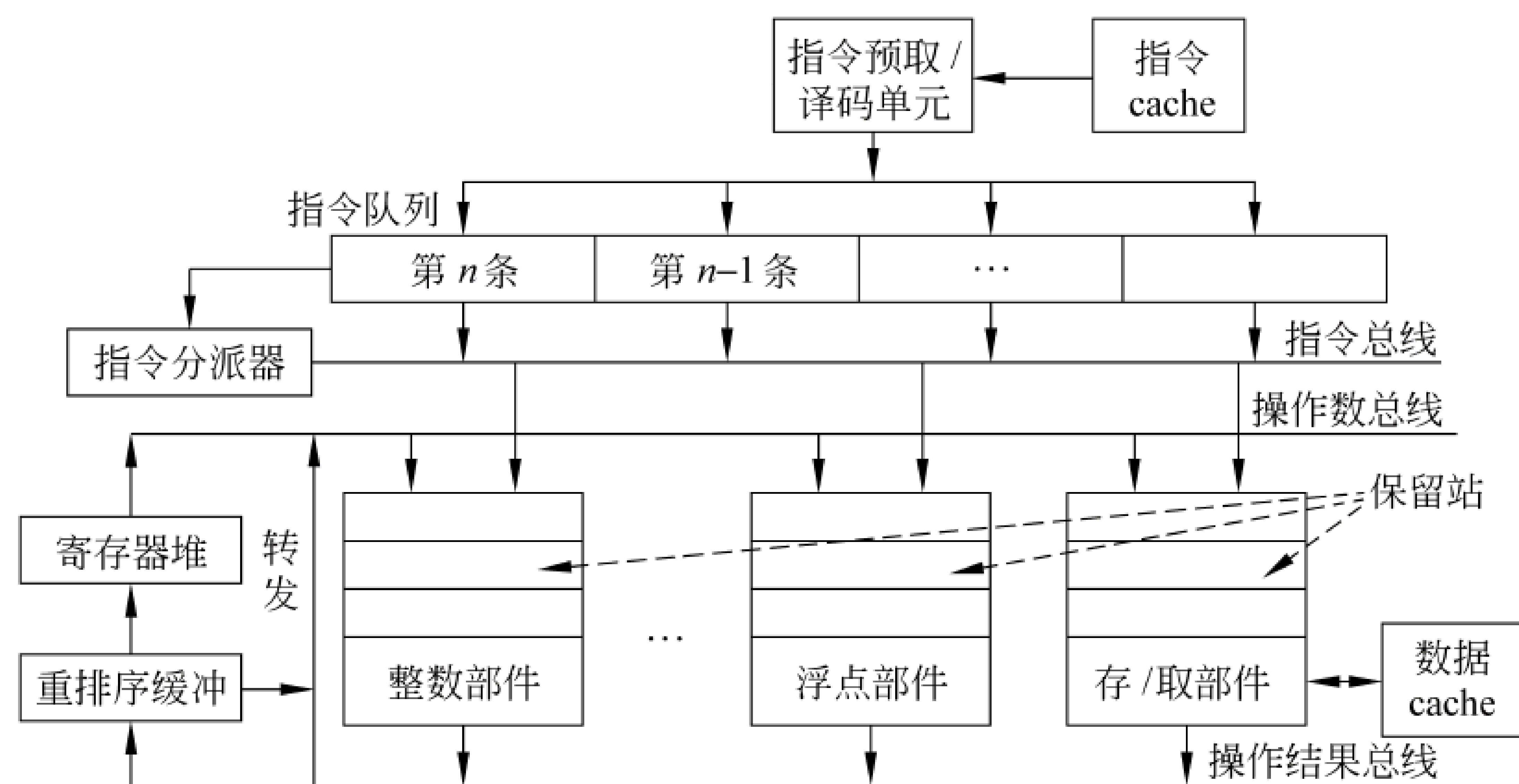


图 7.28 动态多发射流水线处理器的通用模型

(3) 功能单元。超标量处理器中一定有多个功能单元,它们各自完成独立的操作,如整数加、整数乘、整数除、浮点加、浮点乘、浮点除、存数/取数等。每个功能单元都具有一定的操作性能,通常用两个周期数来刻画之。一个是执行周期数(latency),表示完成特定操作所用的时钟周期数;另一个是发射时间(issue time),表示连续、独立的两次操作之间的最短周期数。表 7.1 是 Pentium Ⅲ的部分功能部件性能列表,从表 7.1 可看出,整数加、整数乘、浮点加、取数、存数这 5 种部件是流水化的,浮点乘部件是部分流水化的,而整数除和浮点除是完全没有流水化的。每个功能部件有各自的缓冲器,称为保留站,用于保存操作数和操作命令。

表 7.1 Pentium Ⅲ的功能部件性能

操 作 类 型	执行周期数	发射时间	操 作 类 型	执行周期数	发射时间
整数加法	1	1	浮点数乘法	5	2
整数乘法	4	1	浮点数除法	38	38
整数除法	36	36	取数(cache 命中)	3	1
浮点数加法	3	1	存数(cache 命中)	3	1

(4) 重排序缓冲(ReOrder Buffer)。可简化表示为 ROB,用于保存已完成的指令结果,等待在可能时写回寄存器堆。功能部件一旦完成操作,则将结果同时送其他等待该结果的保留站和重排序缓冲 ROB 中。指令结果也可在 ROB 中被“转发”。当指令发射时,其源操作数可能是其他指令的运算结果,因而可能正在寄存器堆或 ROB 中,此时,可立即将操作数复制到相应的保留站中;若操作数不在寄存器堆或 ROB 中,则一定会在某个时刻由一个功能单元计算出来,硬件通过定位该功能单元,将结果从旁路转发到相应的保留站。

动态多发射流水线的执行模式有三种:按序发射、按序完成;按序发射、无序完成;无序发射、无序完成。下面用一个例子来说明这三种执行模式的实现思想。

例 7.5 假定某个 2-发射超标量处理器的指令执行过程分为取指(IF)、译码(ID)、执行(Ex)、写回(WB)4 个阶段。其中,IF、ID 和 WB 阶段在一个时钟周期内完成,在这三个阶段可同时有两条指令执行。Ex 阶段有三个执行部件:①访存部件用于完成数据 cache 访问,需要一个时钟周期;②整数 ALU 用来完成 ALU 操作,需两个时钟周期;③整数乘法器用来完成乘法运算,需三个时钟周期。整数ALU和乘法器均采用流水化方式执行。假定有一个如图 7.29 所示的指令序列在该处理器上执行,试说明按三种模式(按序发射、按序完成;按序发射、无序完成;无序发射、无序完成)执行指令的过程,并说明各需要多少个时钟周期。

i1	lw	\$1, A
i2	add	\$2, \$2, \$1
i3	add	\$3, \$3, \$4
i4	mul	\$4, \$5, \$4
i5	lw	\$6, B
i6	mul	\$6, \$6, \$7

图 7.29 一个 MIPS 指令序列

解:显然,取指阶段总是按顺序进行,并且可以保证一次取两条指令。因此,下面给出的示意图中只考虑译码、执行和写回三个阶段的情况。根据题意可知,某个时钟周期内,处理器中最多有两条指令在取指、两条指令在译码、一条指令在访存、两条指令进行 ALU 操作、三条指令执行乘法运算、两条指令进行写回操作。也即,最多可有 12 条指令正在被处理。但是这只是理想的情况,实际上,指令序列中指令的排序不一定正好和功能部件一一对应,而且指令之间还存在相互关联,因此,很多情况下,并不是每个功能部件的每个流水段都能被充满。

对图 7.29 所示的指令序列进行分析后可知:指令 i1 和 i2 之间、i3 和 i4 之间、i5 和 i6 之间具有数据相关性,其中 i1 和 i2 之间是 \$1 被写后再读,称为 RAW(Read After Write);i5 和 i6 之间的 \$6 既是 RAW,又是写后写 WAW(Write After Write)。所以不管采用哪种模式,在这两处的两条指令都不能同时发射,并要按序完成。i3 和 i4 之间的 \$4 是读后写 WAR(Write After Read),因此,必须保证在 i3 读 \$4 之前,i4 不能写 \$4。

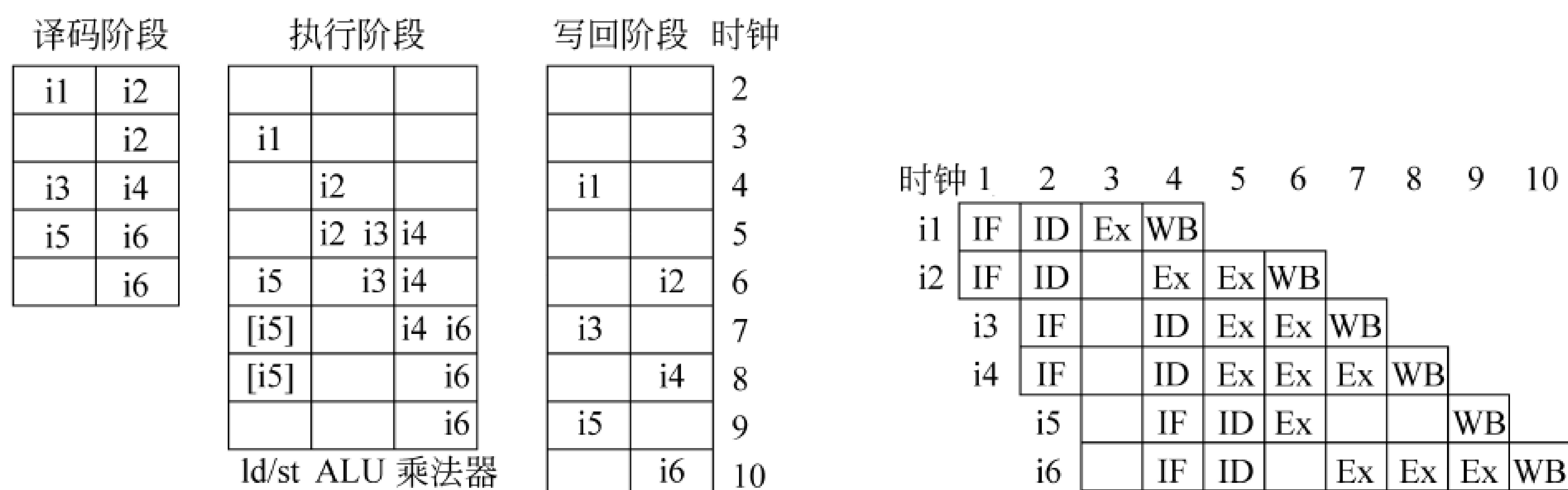
图 7.30(a)、图 7.30(b)和图 7.30(c)分别给出了"按序发射、按序完成"、"按序发射、无序完成"、"无序发射、无序完成"三种指令执行模式下指令序列的执行过程(注:因为一共只有两条乘法指令,故图中乘法部件流水线未充满)。

图 7.30(a)是"按序发射、按序完成"执行模式,从图中可看出,所有阶段都是按顺序进行的。例如,因为在第 3 时钟内 i2 不能和 i1 同时发射,所以,i2 在译码阶段被阻塞一个周期。为了按序完成,虽然 i5 在时钟 6 已经完成,但一直要推迟到 i4 写回后的第 9 时钟才写回。完成上述 6 条指令共需 10 个时钟周期。

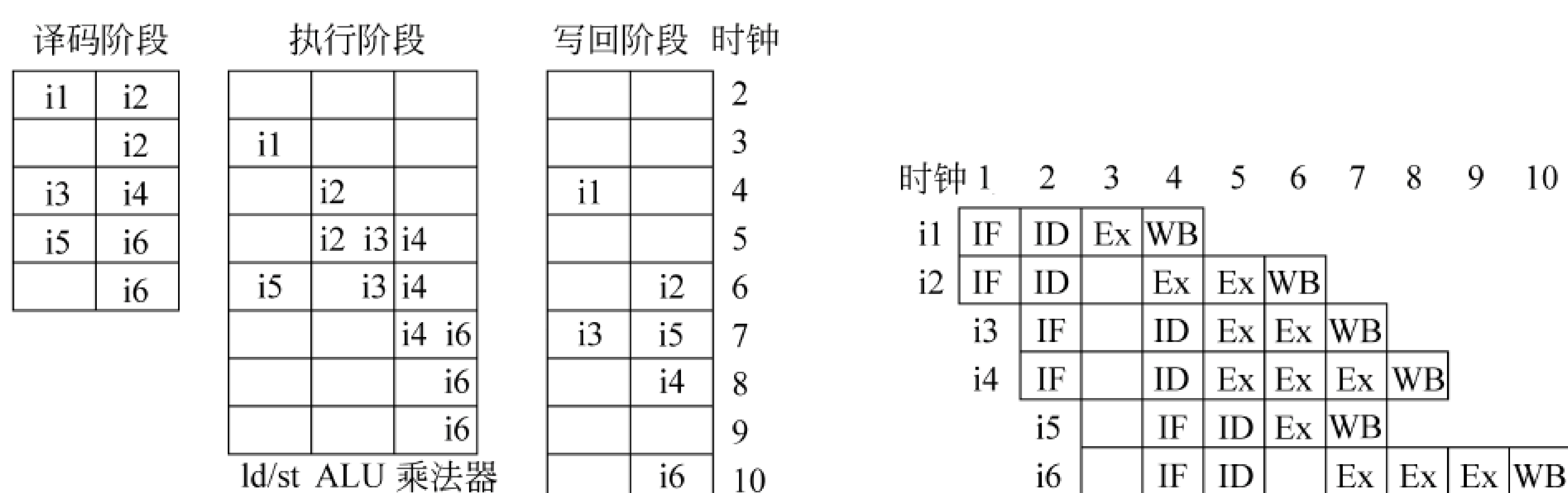
图 7.30(b)是"按序发射、无序完成"执行模式,从图中可看出,在执行阶段,指令的发射是按顺序进行的,而在写回阶段则是无序的。例如,i5 在时钟 6 已经完成,与 i3、i4 没有相关性,所以可先于 i4 写回。这种方式也需 10 个时钟周期完成指令,但访存部件在第 7 和第 8 时钟已空出来,可被其他指令使用。

图 7.30 (c)是"无序发射、无序完成"执行模式,从图中可看出,在执行阶段,指令的发射是无序进行的。在无序发射的超标量处理器中,译码后的指令被存放在一个"指令窗口"缓冲中,等待发射。当所需功能部件可用、并且不会因为冲突或相关性阻碍指令的执行时,就从指令窗口发射,与取指和译码的顺序无关。例如,在时钟周期 4 由于乘法器空闲,所以 i4 在 i3 之前先被发射。在写回阶段指令的完成也是无序的。实际上,只要保证 i1 和 i2 之间、i5 和 i6 之间的发射和完成是按序的即可。这种方式下,因为可以将无关指令提前发射到空

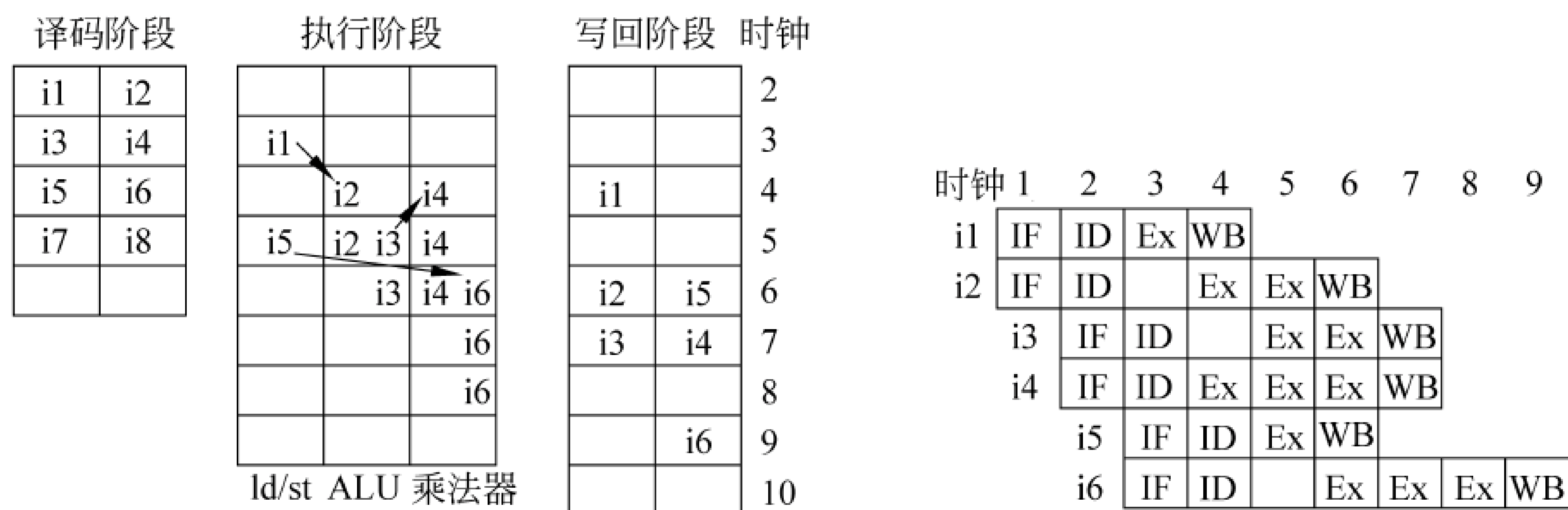
闲部件执行，所以往往能加快程序的执行。此例中，最终只需 9 个时钟周期，比上述两种方式快一个时钟周期。



(a) 按序发射、按序完成



(b) 按序发射、无序完成



(c) 无序发射、无序完成

图 7.30 三种执行模式下的指令序列执行过程

上述例子说明了处理器如何动态调度指令的过程。前面提到，如果编译器先进行静态调度，尽量消除依赖关系，可使流水线达到较高的动态发射速率。例如，对于上述例子给出的指令序列，如果编译器先静态调度指令序列，使 i5 和 i6 调到最前面，并使 i1 和 i2 之间、i3 和 i4之间隔开距离，那么，在"无序发射、无序完成"执行模式下，可以使执行时间缩短为 8 个时钟周期，如图 7.31 所示。

动态调度可在流水线发生阻塞时，动态地提前执行无关指令。前面说过，超标量处理器除了需要编译器进行静态指令调度外，还要依靠处理器进行动态调度。因为并不是所有阻塞都能事先由编译器确定，例如，cache 缺失是编译器无法事先预见的阻塞，只有在动态执行时，才能发现是否发生了 cache 缺失；此外，动态分支预测也需要根据执行的真实情况进

时钟	1	2	3	4	5	6	7	8
i5: lw $6, B	IF	ID	EX	WB				
i6: mul $6, $6, $7	IF	ID		Ex	Ex	Ex	WB	
i1: lw $1, A		IF	ID	Ex	WB			
i3: add $3, $3, $4		IF	ID	Ex	Ex	WB		
i2: add $2, $2, $1			IF	ID	Ex	Ex	WB	
i4: mul $4, $5, $4			IF	ID	Ex	Ex	Ex	WB

图 7.31 静态调度后的无序发射无序完成

行预测。

采用动态调度可使硬件将处理器细节屏蔽起来，不同处理器的发射宽度、流水线延时等可能不同，流水线的结构也会影响循环展开的深度。通过动态调度使得处理器细节屏蔽起来，软件发行商无须针对同一指令集的不同处理器发行相应的编译器，并且，以前的代码也可在新的处理器上运行，无须重新编译。

*7.4.3 Pentium 4 处理器的流水线结构

Pentium 4 是一种“CISC 壳、RISC 核”的体系结构。它利用“踪迹高速缓存（Trace Cache，TC）”来实现指令 cache。在 TC 中存放指令解码后的微操作（μop），每个 μop 相当于一条 RISC 指令。Pentium 4 处理器对 μop 的执行采用了 20 级超流水线技术，使流水线执行效率得到很大提高，但更多的流水线级数使得分支转移预测错误时带来更大的性能损失，因而，Pentium 4 采用了静态和动态两级分支预测，大大提高了预测正确率。图 7.32 是 Pentium 4 处理器的逻辑结构示意图。

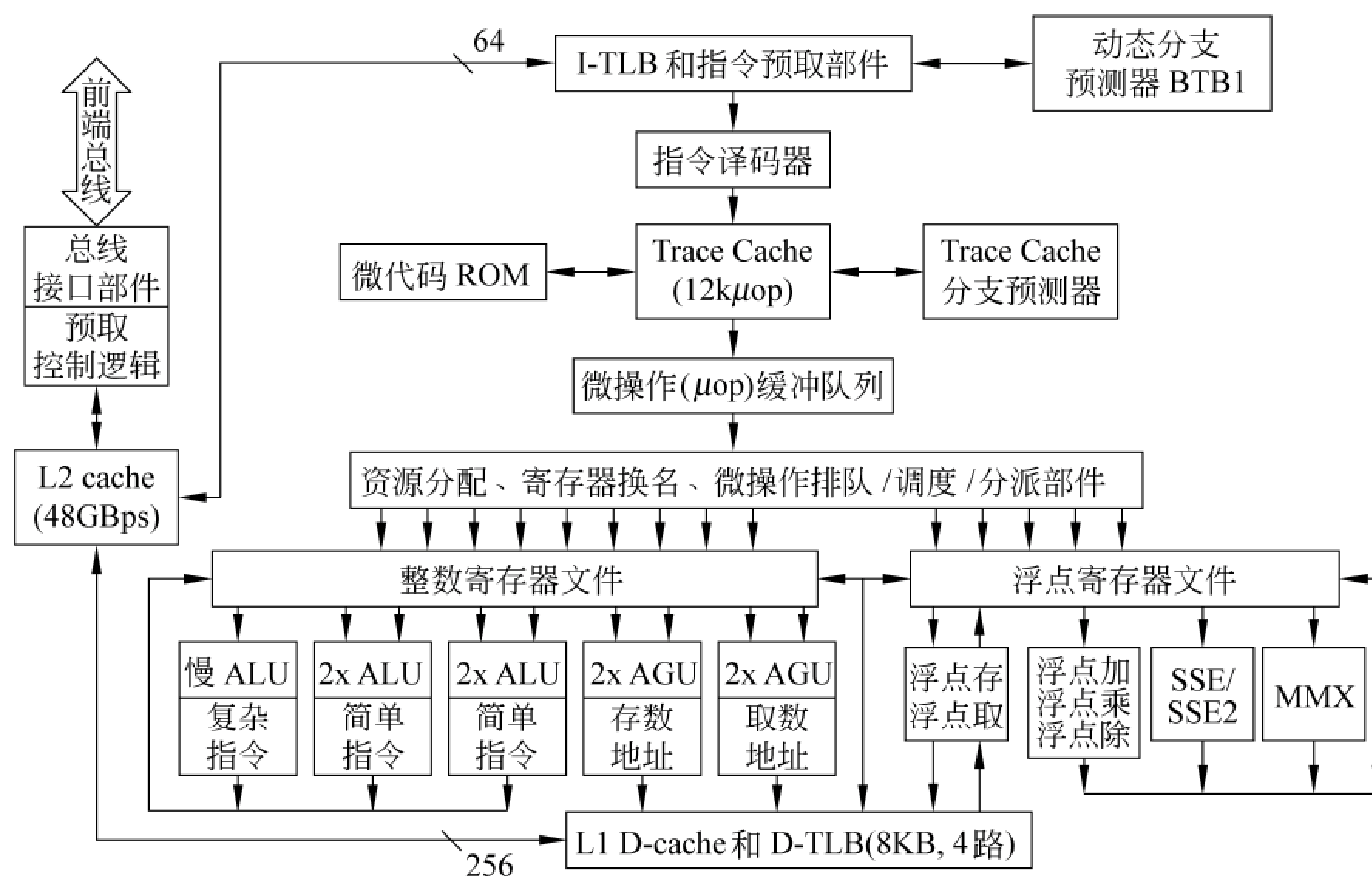


图 7.32 Pentium 4 处理器的逻辑结构

Pentium 4 处理器主要包含两部分：处理器核心结构和 L2 cache。这两部分集成在同一个半导体基片上，并都以主频速度运行。总线接口部件和预取控制逻辑用来实现 L2

cache 通过前端总线与处理器片外进行信息交换。L2 cache 中包含了从主存取来的数据和指令，采用每行 128 字节的 8 路组相联结构，在处理器内部分别和 L1 数据 cache 和指令预取部件交换数据和指令。

处理器核心部分主要包括指令预取部件、指令译码器、Trace Cache、微码 ROM、动态分支预测器 BTB1 和 BTB2、微操作缓冲队列、资源分配/寄存器换名/微操作排队/调度/分派部件、L1 数据 cache、两个 TLB、整数与浮点寄存器文件，以及 9 个功能执行部件。

指令预取部件负责从主存或 L2 cache 取出指令送指令译码器；指令译码器负责将简单指令翻译成一个或多个被称为微操作(μop)的 RISC 指令。转换得到的 μop 被送到 TC 中。对于复杂指令，由于对应的微操作数太多，通过硬件转换比较困难，所以直接送 TC，由 TC 从微码 ROM 中直接取对应的 μop 序列。

当 μop 送到 TC 后，处理器就按“无序发射、无序完成”的 20 级超标量超流水线方式执行，如图 7.33 所示。

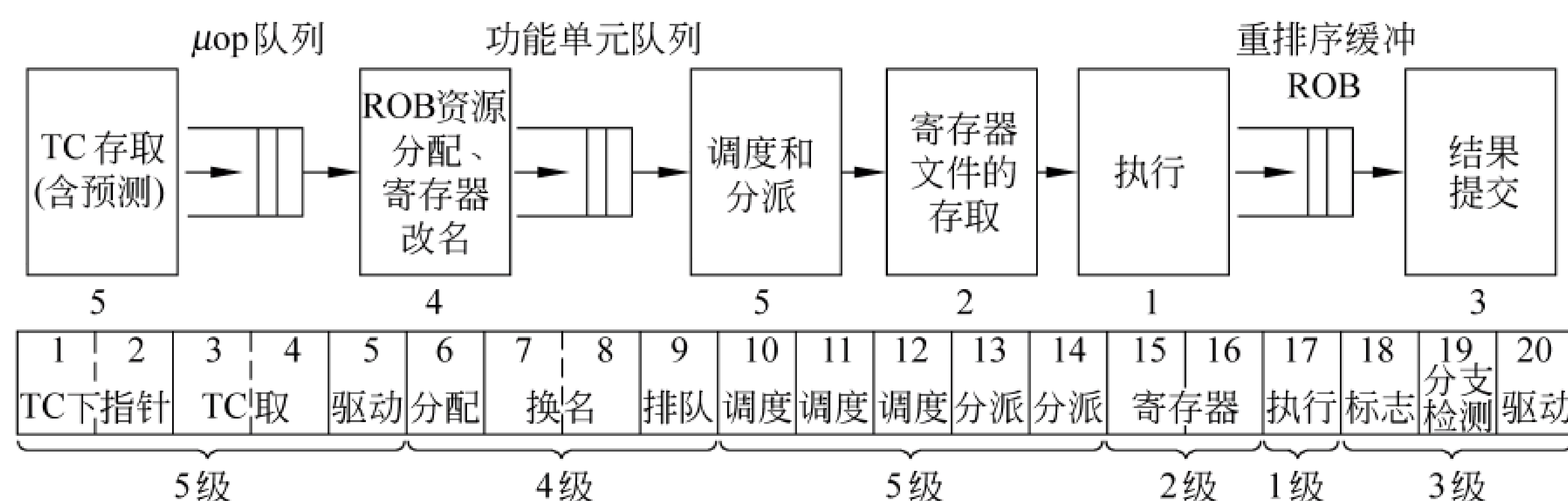


图 7.33　Pentium 4 μop 的 20 级流水线

Pentium 4 的 20 级 μop 流水线分 6 大步骤，分别用 5、4、5、2、1 和 3 个时钟周期完成，共 20 个时钟周期。因为驱动段不完成任何操作，只是用于芯片内传输信号的驱动，使其保证长距离传输，所以，实际功能段只有 11 个，下面分别介绍这些功能段。

(1) TC 下指针(TC next IP)

从 TC 取 μop 的操作使用的是 TC 自身的指针，该段主要用来计算下次从 TC 取 μop 时的指针。经过指令译码器，指令被转换为 μop 序列，源源不断地送到 TC 中，TC 中 μop 按一条条踪迹(Trace)存放，分支 μop 对应的不同分支中一段连续 μop 序列构成一个踪迹。通常沿一个踪迹顺序取，遇到一个分支 μop 时，由动态预测器 BTB2 预测将沿哪条踪迹顺序取。如果被预测到的踪迹不在 TC 时，则通知指令预取器，快从 L2 中取指令并译码。

(2) TC 取(TC Fetch)

根据 TC 指针从 TC 中取出 μop，送到 μop 队列。在取 μop 时，若取到的是从译码器直接送来的未转换为 μop 的复杂指令，则需要到微码 ROM 中取出该复杂指令对应的 μop 序列。通常，一条简单指令最多包含 4 个 μop，而复杂指令则包含 5 个或 5 个以上的 μop。微码 ROM 实际上是一个微操作控制器。

(3) 分配(Allocate)

该段用来为 μop 的执行分配所需的资源，这些资源包括 ROB、物理寄存器或 Load/

Store 缓冲器等,但不包括功能执行单元。ROB 用来记录 μop 的执行状态,共有 126 项,所以最多可以有 126 个 μop 同时在流水线中。一个时钟周期内分配器可同时为三条 μop 分配资源,只要有一个 μop 所需资源不能满足,则分配器延迟操作,直到三个 μop 都满足为止。

(4) 换名(Renaming)

在 Pentium 4 内部,整数和浮点数各有 128 个物理寄存器,寄存器换名操作将用户可见的外部逻辑寄存器换成内部的物理寄存器,换名时,要确定是真实依赖还是名字依赖。名字依赖时,可用不同的物理寄存器替换相同的逻辑寄存器。

(5) μop 排队(Queuing)

有两个队列,一个是存/取队列,用于存储器操作,另一个是整数/浮点数队列,用于除存储器操作以外的其他所有操作。根据资源分配情况进行寄存器换名后,按 FIFO 的次序分别送这两个队列保存,直到被调度出去。因此,队列内部是有序的,但队列之间的 μop 没有顺序关系。

(6) μop 调度(Scheduling)

有 4 个调度器,其中一个存储器操作调度器用于对存/取队列中的 μop 进行调度,其他三个都是为整数/浮点数队列中的 μop 进行调度。调度器必须检测 μop 流的数据相关性,以确定 μop 的先后次序。

(7) 分派(Dispatching)

根据调度结果,将那些没有数据相关性或有相关性但所有源操作数都已就绪的 μop 分派出去,这里的分派是指开始启动读浮点/整数寄存器文件中的内容。源操作数就绪是指对应 ROB 中的状态表明结果已经在寄存器文件中。

(8) 寄存器文件(Register File)

被分派的 μop 在该阶段读取物理寄存器中的源操作数,若源操作数是存储器数据的话,则要从寄存器文件中的旁路网络由 L1-D cache 读取。

(9) 执行(Execute)

将所读出的源操作数送到相应的功能部件执行。每个功能部件也采用流水线方式,因而所需时钟周期数不同。一共有 9 个功能部件,可同时工作。其中有两个高速整数 ALU (每个时钟周期进行两次操作),用于完成简单的整数运算(如加、减法);一个慢速整数 ALU(需要多个时钟周期才能完成一次操作),用于完成整数乘、除法运算;两个地址生成部件(AGU),用于计算操作数的有效地址,所生成的地址分别用于从内存取操作数或向内存保存操作结果;一个运算部件用于完成浮点操作数地址的计算;一个运算部件用于完成浮点加法、乘法和除法运算;一个运算部件用于执行流式 SIMD 处理(SSE/SSE2/SSE3 指令);一个运算部件用于完成多媒体信号处理(MMX 指令)。注意:在运算部件中执行的是微操作,而不是指令。

(10) 标志(Flags)

该段用于建立标志信息(如 ZF/CF 等),并将执行结果写入物理寄存器。

(11) 分支检测(Branch Check)

用于对 BTB2 中的分支预测进行实际确认,并根据确认结果修改 BTB2 中该项的历史

位。若预测不正确，则还要清洗（冲刷）被错误执行的 μop 序列及其执行结果。

7.5 本章小结

本章主要介绍了如何利用流水线来提高指令执行效率。主要内容包括指令流水线的一般原理、流水线方式下的吞吐率和指令执行时间、流水线的局限性、流水线数据通路的设计、流水线的控制信号、流水线寄存器的概念、流水线冒险的种类、结构冒险及其处理、数据冒险及其处理、转发处理、控制冒险及其处理、分支预测原理以及超流水线、超标量和动态流水线等高级流水线的概念，具体总结如下。

- 指令流水线的基本概念
  - 将每条指令的执行规整化为若干个流水阶段。
  - 每个流水阶段的执行时间以最慢的流水段所需时间为准，等于一个时钟周期。
  - 理想情况下，每个时钟有一条指令进入流水线，并有一条指令执行结束。
  - 每个流水段中的部件都是组合逻辑部件，流水线段之间需要加流水段寄存器，组合逻辑中产生的结果在时钟到来时被存储到流水段寄存器中。流水段寄存器用以记录所有流到后面阶段要用的各种信息，例如控制信号、指令、新的 PC 值、参加运算的操作数、指令运算结果、指令异常信息、寄存器读口地址、寄存器写口地址、存储地址等。
  - 指令译码得到的控制信号通过流水段寄存器与本指令的数据信息一起，同步传送到后面各个流水段。
- 指令流水线的局限性
  - 不同指令的功能不同，并不是每条指令都能划分成相同多个阶段，按最复杂指令所需规划流水段后，有些指令的某些流水段执行的可能是空操作。
  - 不同流水阶段的功能不同，并不是每个流水段所用的时间都一样长，按最长时间流水段设置时钟周期后，某些流水段中可能会有时间浪费。
  - 随着流水线深度的增加，流水段寄存器的额外开销比例也增大。
  - 指令在资源冲突、数据相关或控制相关时会发生流水线阻塞，因而影响指令执行效率。
- 指令流水线的执行效率
  - 吞吐率：比非流水线方式下提高若干倍，理想情况下，其倍数为划分的流水段个数。
  - 指令执行时间：由于流水段划分要求的一致性，以及流水段寄存器的额外开销，使得流水线方式下一条指令的执行时间更长了。
- 流水线冒险的种类及其处理基本思想
  - 结构冒险（资源冲突）：多条指令同时要求使用同一个功能部件。
    所用解决策略如下：
    - ▲ 规定每个功能部件在一条指令中只能被使用一次。
    - ▲ 规定每个功能部件只能在某个特定的阶段被使用。
    - ▲ 指令存储器（code cache）和数据存储器（data cache）分开。

- ◆ 数据冒险(数据相关):前面指令的目的操作数是后面指令的源操作数。
  所用解决策略如下:
  - ▲ 用软件(如编译器)在数据相关指令前插入 nop 指令。
  - ▲ 在硬件检测到数据相关时,使后面的数据相关指令进入停顿状态,也即在特定的流水段中插入“气泡”以“阻塞”指令继续执行,直到取得所需数据为止。
  - ▲ 利用“转发(旁路)”技术把前面指令执行过程中得到的数据直接传送到后面指令需要使用数据的地方。
  - ▲ 对于取数后直接使用的情况(如 Load 指令取出的数据是随后下一条运算指令的操作数),则采用“阻塞加转发”的方式解决数据冒险。
- ◆ 控制冒险(控制相关):有两种情况会发生控制冒险。第一种是转移指令引起的。分支、调用、返回、跳转等指令(统称为转移指令)需要计算目标转移地址,分支指令还需根据条件确定新的 PC 值。由于获取转移目标地址的时间较长,使得在目标地址产生前,后续指令已被取到流水线中。如果已经取出并正在执行的指令不是应该执行的指令,则发生了控制冒险。分支指令引起的控制冒险称为分支冒险。第二种是异常和中断引起的。当处理器检测到发生异常和中断时,可能有多条不该执行的指令已在流水线中执行,因而会导致控制冒险。
  所用解决策略如下:
  - ▲ 由编译器在转移指令后面插入 nop 指令(插入指令条数等于延迟损失时间片)。
  - ▲ 在硬件检测到有关转移指令时,使后面若干条指令(指令条数等于延迟损失时间片)进入停顿状态,也即在特定的流水段中插入“气泡”以“阻塞”指令继续执行,直到能确定正确的 PC 值为止。
  - ▲ 对于分支冒险,可结合采用“分支预测”技术。有简单(静态)预测和动态预测两种方式。静态预测方式下,预测的结果是固定的,总是预测转移(taken)或不转移(not taken);动态预测方式下,预测位根据分支指令执行的历史情况进行调整。动态预测比静态预测成功率高得多。
  - ▲ 采用延迟分支技术。将前面几条与转移指令无关的指令放到分支指令后面的延迟槽中执行,无关指令不够时用 nop 指令填满延迟槽。这样,硬件不需对流水线进行阻塞,对指令顺序进行调整的工作在编译阶段完成。

- ● 高级流水线技术。在时间和空间上进一步提高指令级并行性,通常有三种措施。
  - ◆ 超流水线:将指令执行过程划分得更细,采用更多级数的流水线,着重在于时间上并行。
  - ◆ 多发射流水线(超标量处理器):同时发射多条指令,并由多个功能部件并行执行,着重在于空间上并行。“指令打包”和“冒险处理”是实现多发射的两个基本任务。
    - ▲ 静态多发射:“指令打包”和“冒险处理”任务主要由编译器静态完成,打包后的指令相当于一条由多条指令组成的长指令,被同时发射执行,因此,这类处理器有时被称为 VLIW 处理器。Intel 的 IA-64 架构采用这种方法,Intel 称其为 EPIC 技术。
    - ▲ 动态多发射:由处理器硬件在指令执行时动态确定哪些指令被同时发射,如果没有可以被同时发射的指令或遇到指令相关时,则某些流水段空闲。目前超标

量处理器多指采用动态多发射流水线的处理器，并且通常会结合采用动态流水线调度技术。

▲ 动态流水线调度：处理器通过指令相关性检测和动态分支预测等手段，投机性地不按指令顺序执行。即当发生流水线阻塞时，根据指令的依赖关系，动态地到后面找一些没有依赖关系的指令提前执行。在动态流水线调度下，指令被"乱序"执行。

习　题　7

1. 给出以下概念的解释说明。

(1) 指令流水线	(2) 流水线深度	(3) 指令吞吐量	(4) 流水段寄存器
(5) IPC	(6) 流水线冒险	(7) 结构冒险	(8) 数据冒险
(9) 流水线阻塞	(10) 气泡	(11) 空操作	(12) 转发(旁路)
(13) 控制冒险	(14) 分支预测	(15) 动态预测	(16) 延迟时间损失片
(17) 分支延迟槽	(18) 静态多发射	(19) 动态多发射	(20) 超流水线
(21) 超长指令字 VLIW	(22) 超标量流水线	(23) 动态流水线调度	(24) 按序发射
(25) 无序发射	(26) 存储站	(27) 重排序缓冲(ROB)	(28) 乱序执行
(29) 按序完成	(30) 无序完成	(31) 写后读(RAW)相关	(32) 写后写(WAW)相关

2. 简单回答下列问题。

(1) 流水线方式下，一条指令的执行时间缩短了还是加长了？程序的执行时间缩短了还是加长了？

(2) 具有什么特征的指令集易于实现指令流水线？

(3) 流水线处理器中时钟周期如何确定？单条流水线处理器的 CPI 为多少？每个时钟周期一定有一条指令完成吗？为什么？

(4) 流水线处理器的控制器实现方式更类似于单周期控制器还是多周期控制器？为什么？

(5) 为什么要在各流水段之间加寄存器？各流水段寄存器的宽度是否都一样？为什么？

(6) 你能列出哪几种流水线被阻塞的情况？你知道硬件和软件是如何处理它们的吗？

(7) 超流水线和多发射流水线的主要区别是什么？

(8) 静态多发射流水线和动态多发射流水线的主要区别是什么？

(9) 为什么说 Pentium 4 是"CISC 壳、RISC 核"的体系结构？

3. 假定在一个 5 级流水线(如图 7.5 所示)处理器中，各主要功能单元的操作时间如下。存储单元：200ps；ALU 和加法器：150ps；寄存器堆读口或写口：50ps。请回答以下问题。

(1) 若执行阶段 EXE 所用的 ALU 操作时间缩短 20%，则能否加快流水线执行速度？如果能的话，能加快多少？如果不能的话，为什么？

(2) 若 ALU 操作时间增加 20%，对流水线的性能有何影响？

(3) 若 ALU 操作时间增加 40%，对流水线的性能又有何影响？

4. 假定某计算机工程师想设计一个新的 CPU，一个典型程序的核心模块有一百万条指令，每条指令执行时间为 100ps。请回答以下问题。

(1) 在非流水线处理器上执行该程序需要用多长时间？

(2) 若新 CPU 采用 20 级流水线，执行上述同样的程序，理想情况下，它比非流水线处理器快多少？

(3) 实际流水线并不是理想的，流水段之间的数据传送会有额外开销。这些开销是否会影响指令执行时间和指令吞吐率？

5. 假定最复杂的一条指令所用的组合逻辑分成六部分，依次为 A～F，其延迟分别为 80ps、30ps、60ps、

50ps、70ps、10ps。在这些组合逻辑块之间插入必要的流水段寄存器就可实现相应的指令流水线，寄存器延迟为20ps。理想情况下，以下各种方式所得到的时钟周期、指令吞吐率和指令执行时间各是多少？应该在哪里插入流水段寄存器？

(1) 插入一个流水段寄存器，得到一个两级流水线。

(2) 插入两个流水段寄存器，得到一个三级流水线。

(3) 插入三个流水段寄存器，得到一个四级流水线。

(4) 吞吐量最大的流水线。

6. 以下指令序列中，哪些指令对之间发生数据相关？假定采用“取指、译码/取数、执行、访存、写回”5段流水线方式，那么不用“转发”技术的话，需要在发生数据相关的指令前加入几条nop指令才能使这段程序避免数据冒险？如果采用“转发”是否可以完全解决数据冒险？不行的话，需要在发生数据相关的指令前加入几条nop指令才能使这段MIPS程序不发生数据冒险？

```
addu    $s3, $s1, $s0
addu    $t2, $s3, $s3
lw      $t1, 0($t2)
add     $t3, $t1, $t2
```

7. 假定以下MIPS指令序列在图7.18所示的流水线数据通路中执行。

```
addu    $s3, $s1, $s0
subu    $t2, $s3, $s3
lw      $t1, 0($t2)
add     $t3, $t1, $t2
add     $t1, $s4, $s5
```

请回答以下问题。

(1) 上述指令序列中，哪些指令的哪个寄存器需要转发，转发到何处？

(2) 上述指令序列中，是否存在Load-use数据冒险？

(3) 第5周期结束时，各指令执行状态是什么？哪些寄存器的数据正被读出？哪些寄存器将被写入？

8. 假定有一个程序的指令序列为“lw，add，lw，add，…”。add指令仅依赖它前面的lw指令，而lw指令也仅依赖它前面的add指令，寄存器写口和寄存器读口分别在一个时钟周期的前、后半个周期内独立工作。请回答以下问题。

(1) 在带转发的5段流水线中执行该程序，其CPI为多少？

(2) 在不带转发的5段流水线中执行该程序，其CPI为多少？

9. 假定在一个带转发的5段流水线中执行以下MIPS程序段，则怎样调整指令序列使其性能达到最好？

```
lw    $2, 100($6)
add   $2, $2, $3
lw    $3, 200($7)
add   $6, $4, $7
sub   $3, $4, $6
lw    $2, 300($8)
beq   $2, $8, loop
```

10. 在一个采用“取指、译码/取数、执行、访存、写回”的5段流水线中，若检测结果是否为“零”的操作在执行阶段进行，则分支延迟损失时间片（即分支延迟槽）为多少？以下一段MIPS指令序列中，在考虑数

据转发的情况下，哪些指令执行时会发生流水线阻塞？各需要阻塞几个时钟周期？

```
loop: add    $t1, $s3, $s3
      add    $t1, $t1, $t1
      add    $t1, $t1, $s6
      lw     $t0, 0($t1)
      bne    $t0, $s5, Exit
      add    $s3, $s3, $s4
      j      loop
Exit:
```

11. 假设数据通路中各主要功能单元的操作时间如下。存储单元：200ps；ALU和加法器：100ps；寄存器堆读口或写口：50ps。程序中指令的组成比例为：取数25%、存数10%、ALU 52%、分支11%、跳转2%。假设时钟周期取存储单元操作时间的一半，MUX、控制单元、PC、扩展器和传输线路等的延迟都忽略不计，则下面的实现方式中，哪个更快？快多少？

(1) 单周期方式：每条指令在一个固定长度的时钟周期内完成。

(2) 多周期方式：每类指令时钟数为取数-7，存数-6，ALU-5，分支-4，跳转-4。

(3) 流水线方式：取指1、取指2、取数/译码、执行、存取1、存取2、写回7段流水线；没有结构冒险；数据冒险采用“转发”技术处理；Load指令与后续各指令之间存在依赖关系的概率分别为1/2、1/4、1/8、…；分支延迟损失时间片为2，预测准确率为75%；不考虑异常、中断和访问缺失引起的流水线冒险。

12. 有一段程序的核心模块中有5条分支指令，该模块将会被执行成千上万次，在其中一次执行过程中，5条分支指令的实际执行情况如下(T：Taken；N：not Taken)。

分支指令1：T—T—T。

分支指令2：N—N—N—N。

分支指令3：T—N—T—N—T—N。

分支指令4：T—T—T—N—T。

分支指令5：T—T—N—T—T—N—T。

假定各个分支指令在模块每次执行过程中实际执行情况都一样，并且动态预测时，每个分支指令都有自己的预测表项，每次执行该模块时的初始预测位都相同。请分析并给出以下几种预测方案的预测准确率。

(1) 静态预测，总是预测转移（Taken)。

(2) 静态预测，总是预测不转移(not Taken)。

(3) 一位动态预测，初始预测转移（Taken)。

(4) 二位动态预测，初始预测弱转移(Taken)。

第 8 章 系统总线

在计算机系统中，功能部件之间必须互连。部件之间的互连方式有两种，一种是各部件之间通过单独的连线互连，这种方式称为分散连接；另一种是将多个部件连接到一组公共信息传输线上，这种方式称为总线连接。总线连接结构的两个主要优点是灵活和成本低。它的灵活性体现在新部件可以很容易地加到总线上，并且部件可以在使用相同总线的计算机系统之间互换。因为一组单独的连线可被多个部件共享，所以总线的性价比高。总线的主要缺点是它可能产生通信瓶颈。现代计算机普遍使用的是总线互连结构。

本章着重介绍总线的基本概念、总线设计中的几个要素、总线标准以及现代计算机的总线互连结构。

8.1 总线的基本概念

计算机由 CPU、存储器和 I/O 模块组成，而计算机的所有功能都是通过 CPU 执行指令实现的。在指令执行过程中，CPU、存储器和 I/O 模块之间要不断地交换数据(包括指令、中断向量等)，因此，可以说计算机所有功能的实现，归根结底是各种信息在计算机内部的各部件之间进行交换的过程。要进行信息交换，必须在部件之间构建通信线路，通常把连接各部件的通路的集合称为互连结构。早期的计算机大多采用分散连接方式，相互通信的部件之间都有单独的连线，如果某个部件与其他所有部件都有信息交换的话，它的内部连线就非常复杂，而且成本高。此外，某一时刻它只能和其他部件中的一个交换信息，这将大大影响系统的工作效率。特别是随着计算机应用领域的不断扩大，I/O 设备的种类和数量越来越多，用分散方式无法满足人们随时增减设备的需要，因此出现了总线连接方式。

总线是连接两个或多个功能部件的一组共享的信息传输线，它的主要特征就是多个部件共享传输介质。一个部件发出的信号可以被连接到总线上的其他所有部件所接收。

*8.1.1 总线的特性和分类

总线是连接多个部件的信息传输线，是各部件共享的传输介质，因此，必须规定一些基本特性。

(1) 物理特性。总线的物理特性是指总线在物理连接上的特性，包括连线类型、数量、接插件的几何尺寸和形状以及引脚线的排列等。

从连线的类型来看，总线可分为电缆式、主板式和底板式。电缆式总线通常采用扁平电缆连接电路板；主板式总线通常在印刷电路板或卡上刻蚀出平行的金属线，这些金属线按照某种排列以一组连接点的方式提供插槽，系统的一些主要部件借助于这些插槽连接到系统总线，例如，主板上的前端总线、存储器总线等；底板式总线(backplane bus)则通常在机箱中设置插槽板，其他功能模块一般都做成一块卡，通过将卡插到槽内连接到系统中，例如 PCI 总线、AGP 总线等。

从连线的数量来看，总线一般分为串行总线和并行总线。在并行传输总线中，按数据线的宽度分为 8 位、16 位、32 位、64 位总线等。总线的宽度对实现的成本、可靠性和数据传输率的影响较大。

(2) 电气特性。总线的电气特性是指总线的每一条信号线的信号传递方向、信号的有效电平范围等特性。若规定由 CPU 发出的信号为输出信号，送入 CPU 的信号为输入信号，则地址线一般为输出信号，数据线为双向信号，控制线的每一根都是单向的，有的为输出信号，有的为输入信号。总线的电平表示方式有两种：单端方式和差分方式。在单端电平方式中，用一条信号线和一条公共接地线来传递信号。信号线中一般用高电平表示逻辑“1”，低电平表示逻辑“0”。差分电平方式采用一条信号线和一个参考电压比较来互补传输信号，因此一般采用负逻辑，即用高电平表示逻辑“0”，低电平表示逻辑“1”。例如，串行总线接口标准 RS-232C，其电气特性规定低电平要低于－3V，表示逻辑“1”；高电平要高于＋3V，表示逻辑“0”。

(3) 功能特性。总线功能特性是指总线中每根传输线的功能。如地址线用来传输地址信息，数据线用来传输数据信息，控制线用来发出控制信息，不同的控制线其功能不同。

(4) 时间特性。总线时间特性是指总线中任一根传输线在什么时间内有效以及每根线产生的信号之间的时序关系。时间特性一般可用信号时序图来说明。

计算机系统中有多种总线，它们在各个层次上提供部件之间的连接和信息交换通路。根据所连接部件的不同，总线通常被分成三种类型：内部总线、系统总线和通信总线。

(1) 内部总线。指芯片内部连接各元件的总线。例如，CPU 芯片内部在各个寄存器、ALU、指令部件等各元件之间互连的总线。

(2) 系统总线。指连接 CPU、存储器和各种 I/O 模块等主要部件的总线。由于这些部件通常都制作在插件板卡上，所以连接这些部件的总线一般是主板式或底板式总线，主板式总线是一种板级总线，主要连接主机系统印刷电路板中的 CPU 和主存等部件，因此也被称为处理器-主存总线，有的系统将处理器总线和存储器总线分开，中间通过桥接器连接，CPU 芯片通过 CPU 插座插在处理器总线上，内存条通过内存条插座插在存储器总线上。底板式总线通常用于将系统中的各个 I/O 功能模块连接到主机，实现 I/O 设备和主机的连接，所以，底板式总线属于 I/O 总线，典型的有 PCI 总线、AGP 总线、PCI-Express 总线等。

(3) 通信总线。这类总线用于主机和 I/O 设备之间或计算机系统之间的通信。由于这类连接涉及到许多方面，包括设备类型、距离远近、速度快慢和工作方式等，差异很大，所以通信总线的种类很多。

8.1.2 系统总线的组成

系统总线通常由一组控制线、一组数据线和一组地址线构成。也有些总线没有单独的

地址线，地址信息通过数据线来传送，这种情况称为数据线和地址线复用。

数据线用来承载在源部件和目的部件之间传输的信息，这个信息可能是数据、命令或地址（如果数据线和地址线复用的话）。数据线的条数被称为总线宽度，它决定了每次能同时传输的信息的位数。因此数据总线的宽度是决定系统总体性能的关键因素之一。

地址线用来给出源数据或目的数据所在的主存单元或 I/O 端口的地址。例如，若要从主存单元读一个数据，那么 CPU 必须将该主存单元的地址送到地址线上。若要输出一个数据到外设，那么，CPU 也必须把该外设的地址（实际上是 I/O 端口的地址）送到地址线上。地址线是单向的，它的位数决定了可寻址的地址空间的大小。例如，若地址线有 16 位，不采用分时多次传送地址的话，则可访问的存储单元最多只能有 2^{16} 个。

控制线用来控制对数据线和地址线的访问和使用。因为数据线和地址线是被连接在其上的所有设备共享的，如何使各个部件在需要时使用总线，需靠控制线来协调。控制线用来传输定时信号和命令信息。

典型的控制信号（或控制信号组合后的含义）包括以下几个。

- 时钟（Clock）：用于总线同步。
- 复位（Reset）：初始化所有设备。
- 总线请求（Bus Request）：表明发出该请求信号的设备要使用总线。
- 总线允许（Bus Grant）：表明接收到该允许信号的设备可以使用总线。
- 中断请求（Interrupt Request）：表明某个中断源正在请求。
- 中断回答（Interrupt Response）：表明某个中断请求已被接受。
- 存储器读（Memory Read）：从指定的主存单元中读数据到数据总线上。
- 存储器写（Memory Write）：将数据总线上的数据写到指定的主存单元中。
- I/O 读（I/O Read）：从指定的 I/O 端口中读数据到数据总线上。
- I/O 写（I/O Write）：将数据总线上的数据写到指定的 I/O 端口中。
- 传输确认（Transfer Acknowledgement）：表示数据已被接收或已被送到总线上。

8.2　总线设计的要素

尽管有许多不同的总线实现方式，但总线设计的基本要素和考察的性能指标都是一样的。总线设计时要考虑的基本要素包括信号线类型、事务类型、总线带宽、仲裁方法和定时方式等。

8.2.1　信号线类型

总线的信号线类型有专用和复用两种。专用信号线就是信号线专门用来传送某一种信息。例如，使用分立的数据线和地址线分别专门用来传送数据和地址。复用信号线就是指用同一种信号线在不同的时间传输不同的信息。例如，可采用数据/地址线分时复用方式，用一组数据线在总线事务的地址阶段传送地址信息，在数据阶段传送数据信息。这样就使得地址和数据通过同一组数据线进行传输。

信号线的分时复用，可以使用较少的信号线传输更多的信息，从而节省空间和成本。但挂接的每个部件的电路变得更复杂了。而且，因为限定了共享同一组信号线的事件不能同

时发生，所以它还潜在地降低了性能。例如，“存储器写”总线事务中，如果采用专用的数据线和地址线的话，主存单元地址和数据就可以同时送到总线上，而在数据/地址线分时复用的情况下就不能这样。

8.2.2 总线事务类型

通常把在总线上一对设备之间的一次信息交换过程称为一个“总线事务”，把发出事务请求的部件称为主控设备，也称请求代理，另一个部件称为从设备，也称响应代理。例如，处理器要求读取存储器中某单元的数据，则处理器是主控设备(请求代理)，而存储器是从设备(响应代理)。总线事务类型通常根据它的操作性质来定义。典型的总线事务类型有“存储器读”、“存储器写”、“I/O 读”、“I/O 写”、“中断响应”等，一次总线事务简单来说包括两个阶段：地址阶段和数据阶段。

突发(burst)传送事务由一个地址阶段和多个数据阶段构成，用于传送多个连续单元的数据，地址阶段送出的是连续区域的首地址。因此，一次突发传送事务可以传送多个数据，也称为成组传送事务。

不同总线规定的事务类型不一样，通常一个总线可以有多种不同的事务类型，不同事务类型的总线传输过程不同。例如，Pentium Pro 处理器总线的主要事务类型有以下 11 种。

- 延迟回答：当从设备需要花很长时间才能完成某个操作时，就通过发出“延迟回答”事务以“分离事务”方式来处理该事务。这种情况下，需要对主控设备进行寻址，以便将被延迟事务的完成状态以及所读数据(如果是读事务的话)发送给主控设备。
- 中断响应：当处理器响应从 8259 中断控制器送来的中断请求而要去读中断向量时，处理器就发出“中断响应”事务。
- 特殊事务：当处理器广播一条与某内部事件(如 shutdown、halt 等)有关的消息时，就产生“特殊事务”。
- 分支跟踪消息：在指令执行过程中当一个分支指令的条件满足时，处理器就生成“分支跟踪消息”事务。它将送出转移指令的地址和转移到的目标指令的地址。
- I/O 读：当执行输入指令 IN 或 INS 要从某 I/O 设备读取数据或状态时，处理器发出“I/O 读”事务。
- I/O 写：当执行输出指令 OUT 或 OUTS 将数据或命令写到某 I/O 设备时，处理器发出“I/O 写”事务。
- 存储器读并无效：处理器产生“存储器读并无效”事务，主要是为了对一个主存块进行独占访问。
- 存储器代码读：当处理器要从存储器中取指令时产生“存储器代码读”事务。
- 存储器数据读：当某指令需要从存储器中取数据时，产生“存储器数据读”事务。
- 存储器写(不可重试)：当处理器要写回一个被更新过的主存块到存储器以便为一个新的主存块腾出一个 cache 行时，由处理器发出该事务。
- 存储器写(可重试)：当某指令需写数据到存储器时，由处理器发出该事务。

Pentium Pro 处理器总线的每个总线事务包含以下 5 个操作阶段：请求(地址)阶段、检错阶段、侦听阶段、响应阶段和数据阶段。

- 请求(地址)阶段：送出地址、事务类型及有关事务的其他信息。

- 检错阶段：对请求阶段送出的地址和请求事务信息进行奇偶校验检测。
- 侦听阶段：对请求阶段送出的地址检查其在 cache 的命中情况，以确定以后阶段该如何处理该事务。
- 响应阶段：根据请求的事务类型、检错和侦听结果，确定如何响应当前事务。返回的响应结果可以是重试事务、延迟事务、硬件错、无数据传送事务、回写（write back）事务或正常数据传送事务等。对于前 4 种情况，事务在响应阶段就可结束。
- 数据阶段：送数据到数据总线上，或从数据总线上取数据。

8.2.3 总线带宽

总线带宽指总线的最大数据传输率，即总线在进行数据传输时单位时间内最多可传输的数据量，不考虑其他如总线裁决、地址传送等所用的时间。

对于同步总线，其总线带宽的计算公式为：

$$B=W\times F/N$$

其中，W 为总线宽度，即总线能同时并行传送的数据位数，通常以字节为单位；F 为总线的时钟频率；N 为完成一次数据传送所用的时钟周期数。

例 8.1 假定某同步总线在一个时钟周期内传送一个 4 字节的数据，总线时钟频率为 33MHz，求总线带宽是多少？如果总线宽度改为 64 位，一个时钟周期能传送两次数据，总线时钟频率为 66MHz，则总线带宽为多少？提高了多少倍？

解：由上述同步总线带宽计算公式，可得总线带宽为 4B×33MHz/1=132MBps。

总线性能改进后的总线带宽为 8B×66MHz/0.5=1056MBps，提高了 8 倍。

8.2.4 总线裁决

总线是共享的传输介质，一组总线被挂接在其上的所有设备共享，因此，某一时刻只能有一对设备使用总线进行数据传输。当多个设备需要使用总线进行通信时，每个设备为各自的传输都试图将信号送到总线上，如果没有任何控制，就会产生冲突。

这种冲突可以通过引入一个或多个总线主控设备而加以避免。总线上连接的各个部件，根据其对总线有无控制能力被分为主控设备和从设备两种。总线主控设备控制对总线的访问，它能够发起并控制所有总线请求。从设备只能响应从主控设备发来的总线命令，每个总线事务总是在一对主、从设备之间进行数据传送。例如，对于处理器通过总线访问存储器的操作来说，一定是处理器发起一个对存储器访问的总线请求，所以处理器总是一个总线主控设备，而存储器总是一个从设备，因为它只能响应读或写请求，不会产生请求。

最简单的计算机系统只有一个单总线，而且总线中只有一个主控设备：处理器。在一个单主控设备系统中，所有总线操作都必须由处理器控制，所以无须总线裁决。这种方式的主要缺点是处理器必须介入到每个总线事务中。

在一个多主控设备的总线中，每个主控设备都能启动数据传送。因此，必须提供一种机制用来决定在某个时刻由哪个设备拥有总线使用权。决定哪个主控设备能得到总线使用权的过程称为总线裁决。

总线裁决有多种方案。粗略来分，有两类总线裁决方式：集中式和分布式。集中式是将控制逻辑做在一个专门的总线控制器或总线裁决器中，使所有的总线请求集中起来，利用

一个特定的裁决算法进行裁决。而在分布式的裁决方式中，则没有专门的总线控制器，其控制逻辑分散在各个部件或设备中。

主控设备使用总线请求信号进行总线请求，获得总线控制权后，设备就可以使用总线。使用完总线后，设备通过某种方式向裁决器发出信号，表示不再需要总线，此时裁决器将总线再分配给另一个设备。大多数具有多主控设备的总线都有一组连线来传送总线请求和总线许可信号。有时用于总线裁决的信号线在物理上是单独的，有时则采用信号线复用的方式，例如，可以利用数据线进行总线请求。但是，这种情况下，总线裁决和数据传输不能同时进行。

在选择哪个设备获得总线使用权时，一般的裁决方案通常需要考虑以下两个因素的平衡：第一是“等级性”，即每个主控设备有一个总线优先级，具有最高优先级的设备应该先被服务；第二是“公平性”，即任何设备，即使是具有最低优先权的设备也不能永远得不到总线使用权。这种“公平性”保证了想使用总线的每个设备最终总能得到总线。除了上述因素外，更复杂的方案还要考虑怎样缩短总线裁决时间。因为裁决时间是一个额外开销，它增加了总线响应时间和总线访问时间，应尽量缩短并尽可能与总线传送操作在时间上重叠。

1. 集中裁决方式

常用的集中裁决方式有三种：链式查询、计数器定时查询和独立请求，裁决方式如图 8.1 所示。

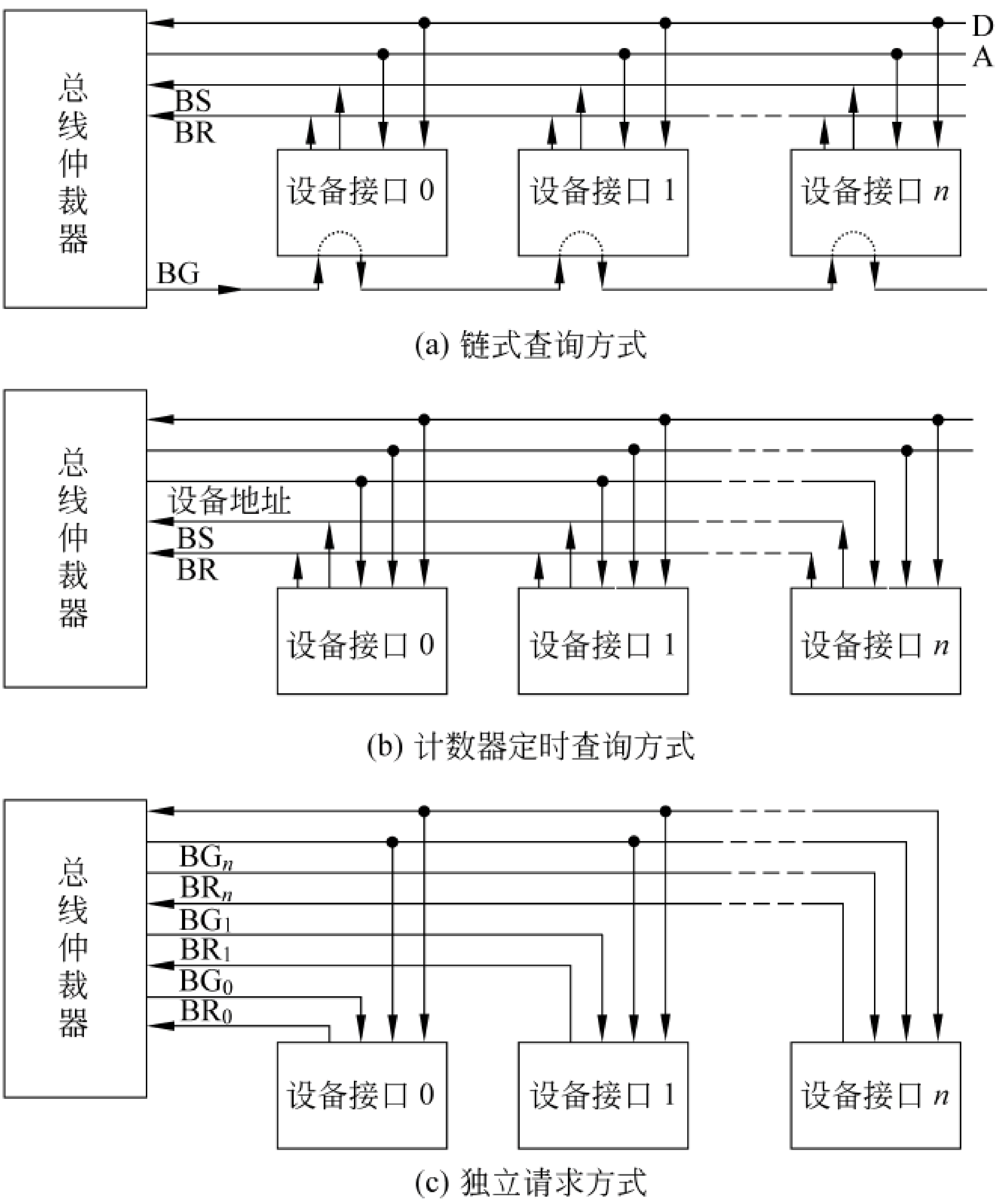


图 8.1　集中式总线仲裁方法

(1) 链式查询方式

链式查询方式也称菊花链查询方式，其裁决过程如图 8.1(a)所示，图中 D 表示双向的数据线，A 表示单向的地址线。另外有三根控制线用于总线控制：BS-总线忙、BR-总线请求、BG-总线允许。主控设备的总线请求信号通过“线或”方式被送到总线裁决器。总线裁决器接收到 BR 送来的总线请求信号后，在总线空闲(即 BS＝0)的情况下，将总线允许信号 BG 通过总线允许线从最高优先权的设备依次向最低优先权的设备串行传送。如果 BG 信号到达的设备有总线请求，则 BG 信号不再往下传，并建立总线忙 BS 信号，表示它已获得了总线使用权，下一个总线事务开始该设备就可控制总线。

在这种方案中，优先级由主控设备在总线上的位置来决定，要求拥有总线使用权的高优先级设备简单地拦截总线允许信号，不让其他更低级的设备收到该信号。

菊花链查询总线简单，只需很少几根线就能按一定优先次序实现总线裁决，而且扩充设备较简单。缺点是不能保证公正性，也即一个低优先级请求可能永远得不到允许，此外，对电路故障也较敏感，因为 BG 信号采用串行传递方式，所以前面设备的故障会影响 BG 信号向后面设备的传递，另外，菊花链的使用也限制了总线速度，因而，其裁决速度较慢。

(2) 计数器定时查询方式

计数器定时查询方式如图 8.1(b)所示，它比链式查询方式多了一组设备地址线，少了一根总线允许线 BG。总线裁决器接收到 BR 送来的总线请求信号后，在总线空闲的情况下，由计数器开始计数，并将计数值通过设备地址线向各设备发出。当某个发出总线请求的设备的编号与从设备地址线上取得的计数值一致时，该设备便获得总线使用权，此时终止计数查询，同时该设备建立总线忙 BS 信号。

计数器的初始值可由程序来设置，因而设备的优先级可以通过设置不同的计数初始值来改变。若每次计数总是从 0 开始，则设备的优先次序是固定的；若每次计数的初始值总是上次得到控制权的设备随后设备的编号，那么所有设备的优先级就是平等的，是一种循环优先级方式。计数器定时查询方式除了具有优先级灵活这个优点外，它对电路故障也不像菊花链查询那样敏感。但是，这种方式增加了一组设备地址线，并且，每个设备要对设备地址进行译码处理，因而控制也复杂一些。

(3) 独立请求方式

独立请求方式如图 8.1(c)所示，每个设备都有一对总线请求线 BR_i 和总线允许线 BG_i，各个设备独立请求总线。当某个设备要求使用总线时，就通过其对应的总线请求线将请求信号送到总线裁决器。总线裁决器中有一个判优电路，可根据各个设备的优先级确定选择哪个设备使用总线。裁决器可以给各个请求线以固定的优先级，也可以通过编程方式设置优先级。

独立请求方式的优点是，响应速度快；如果是可编程的话，优先级还可灵活设置。但它的控制逻辑较复杂，控制线数量比其他两种方式多。例如，若 n 表示允许挂接的最大主控设备数，则菊花链查询方式只需两根裁决线，计数查询方式大致需用 $\log_2 n$ 根裁决线，而独立请求方式则需要 $2n$ 根裁决线。

独立请求方式中的裁决算法在总线裁决器中由硬件来实现，可以采用固定的并行判优算法、平等的循环菊花链算法、动态优先级算法(如最近最少用算法、先来先服务算法)等。

有些总线将查询方式和独立请求方式结合起来。例如，VME 总线使用了多个菊花链，每个菊花链具有一对独立的请求线和允许线，一个并行判优的优先权编码器从多个请求线中选择。

***2. 分布式裁决方式**

常用的分布裁决方式有三种：自举分布式、冲突检测分布式和并行竞争分布式。

(1) 自举分布式裁决

自举分布式裁决过程如图8.2所示，图中假定总线上有4个设备，每个设备有一个唯一的标识号i(从0到3)，其中设备0的优先级最低，设备3的优先级最高。BR_0为总线忙信号线，BR_i(i从1到3)为设备i的总线请求线。设备0在确认BR_0上没有有效信号(即总线空闲)且BR_1、BR_2和BR_3上没有其他设备发出的请求信号时，它就使用总线，将BR_0信号线置为有效，表示总线将处于忙状态；设备1在确认总线空闲且BR_2和BR_3上没有请求信号时能使用总线；设备2在确认总线空闲且BR_3上没有请求信号时就能使用总线；而设备3在总线空闲时可直接使用总线。由此可见，优先级最低的设备0可以不需要总线请求线，因为没有其他设备需要它的请求信号。

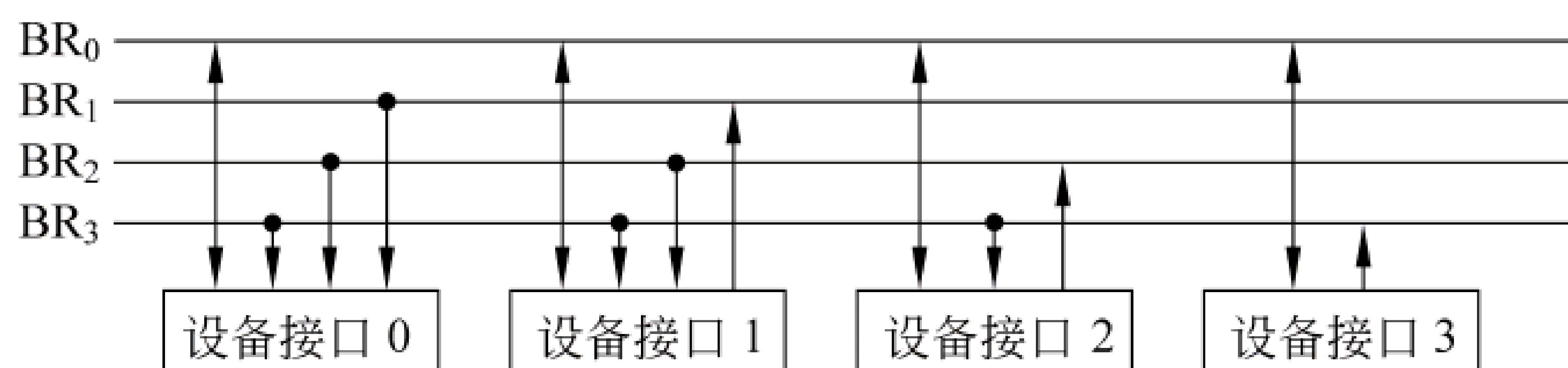


图8.2　自举分布式裁决

这种方案不需要集中的总线裁决器，每个设备独立决定自己是否是最高优先级的请求者，并且优先级固定，每个需要请求总线控制权的设备在各自对应的总线请求线上送出请求信号，在总线裁决期间，每个设备通过取回的信息能够检测出其他比本设备优先级高的设备是否发出了总线请求，如果没有，则立即使用总线，并通过总线忙信号阻止其他设备使用总线；如果一个设备在发出总线请求的同时，检测到其他优先级更高的设备也请求使用总线，则本设备不能马上使用总线。

由于本方案需要较多的连线用于请求信号，所以，许多采用这种方案的总线用数据线DB作为总线请求线。早期MacintoshⅡ中的底板式总线NuBus就采用了该方案，SCSI总线接口也采用了该方案。

(2) 冲突检测分布式裁决

当某个设备要使用总线时，它首先检查是否有其他设备正在使用总线，如果没有，那它就置总线忙，然后使用总线；若两个设备同时检测到总线空闲，那它们可能都会立即使用总线，此时，便发生了冲突。一个设备在传输过程中，它会侦听总线以检测是否发生了冲突，当发现冲突时，两个设备都会停止传输，延迟一个随机时间后再重新使用总线。就像两个有礼貌的人发现自己和对方同时走上一个独木桥后，两人又都同时退回一样，过一个随机时间段后，就可能有一人先走，这样冲突就解决了。这种方案一般用在网络通信总线上，例如，以太网就使用该方案进行总线裁决。

(3) 并行竞争分布式裁决

这是一种较复杂但有效的裁决方案。其基本思想是总线上的每个设备都有一个唯一的仲裁号，需要使用总线的主控设备把自己的仲裁号发送到仲裁线上，这个仲裁号将用在并行竞争算法中。每个设备根据仲裁算法决定在一定的时间段后占用总线还是撤销仲裁号。

并行竞争分布式裁决逻辑如图8.3所示，虚框内是每个设备的仲裁逻辑，有多个设备挂

接在总线上。设备和总线采用相反的逻辑,设备采用正逻辑,而总线则采用负逻辑。假定总线中有8根仲裁线AB0～AB7,所有需要使用总线的主控设备把自己的仲裁号cn0～cn7(相当于设备号)发送到这8根仲裁线上,根据并行竞争算法,发送最大仲裁号的设备将获得总线使用权。

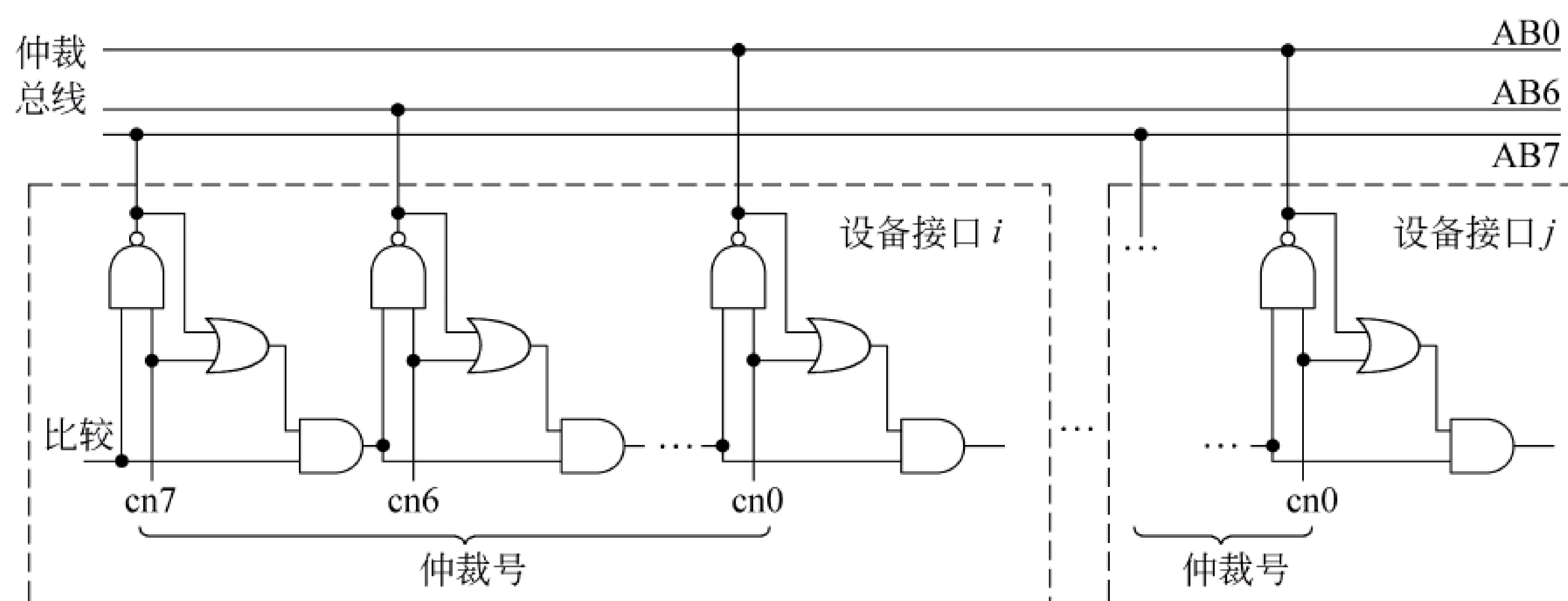


图 8.3　并行竞争分布式裁决

裁决过程如下。仲裁线逻辑是“线或”,因此,只要有一个设备把逻辑“1”送到某个特定的仲裁线上,那么这根线的信号就是“1”,即低电平。每个设备中的裁决逻辑检测仲裁线上的结果,并根据下列规则修改它放到总线上的仲裁号:如果该设备的仲裁号中有某一位为0,而这一位对应的仲裁线信号为“1”,则修改这个仲裁号,使其所有低位都从总线上撤销,也即使所有低位对应的仲裁线送出一个为“0”的信号。这样,具有最高仲裁号的设备将会发现它的仲裁号和留在仲裁线上的号匹配,所以它将赢得总线使用权。

下面用一个例子来说明裁决逻辑的工作原理。假定总线上同时有两个设备要求使用总线,它们的仲裁号分别是00000101和00001010。对裁决逻辑电路进行分析,可知,裁决号1中从左边开始的第5位0(即cn3=0),和对应裁决线AB3上的“1”(低电平),一起被送到一个“或”门,使得或门输出为0,然后这个“0”通过与门被送到后面所有的裁决位,使得裁决号1后面的所有位都被修改为“0”,对应的裁决线上都是高电平。因而,最终留在仲裁线上的号为00001010,即仲裁号大的设备赢得总线使用权,表8.1给出了分析结果。

表 8.1　并行竞争裁决逻辑举例分析结果

裁决号 1		裁决号 2		裁决线电平	裁决线逻辑值
cn	AB	cn	AB		
0	高	0	高	高	0
0	高	0	高	高	0
0	高	0	高	高	0
0	高	0	高	高	0
0	高	1	低	低	1
1	高	0	高	高	0
0	高	1	低	低	1
1	高	0	高	高	0

这种方式与自举分布式裁决算法比，它可以用很少的裁决线挂接大量的设备。例如，假定是8位仲裁号，自举分布式裁决只能表示8个优先级，而这种方式可以表示256个优先级，仲裁号为255的设备优先级最高，而0最低。Futurebus＋总线标准使用这种裁决方案。

8.2.5 定时方式

通过总线裁决确定了哪个设备可以使用总线，那么一个取得了总线控制权的设备如何控制总线进行总线操作呢？也即，如何来定义总线事务中的每一步何时开始、何时结束呢？这就是总线通信的定时问题。总线通信的定时方式有4种：同步、异步、半同步和分离事务。

1. 同步通信方式

同步总线采用公共的时钟信号进行定时，挂接在总线上的所有设备都从一个公共的时钟线上获得定时信号。一定频率的时钟信号定义了等间隔的时间段，这个固定的时间段为一个总线时钟周期。

同步总线的传输协议非常简单，只要在规定的第几个时钟周期内完成特定的操作即可。例如，对于处理器通过总线访问存储器的操作来说，可以规定以下"存储器读操作"协议：主控设备（即处理器）在第一个时钟周期发送存储地址和存储器读命令（可利用控制线表明请求的类型为"存储器读"），从设备（即存储器）总是在第5个时钟周期将数据放到总线上作为响应，处理器也在第5个时钟周期从数据线上取数据。这个读操作的同步时序如图8.4所示。"存储器写"操作的同步时序类似于"存储器读"操作。

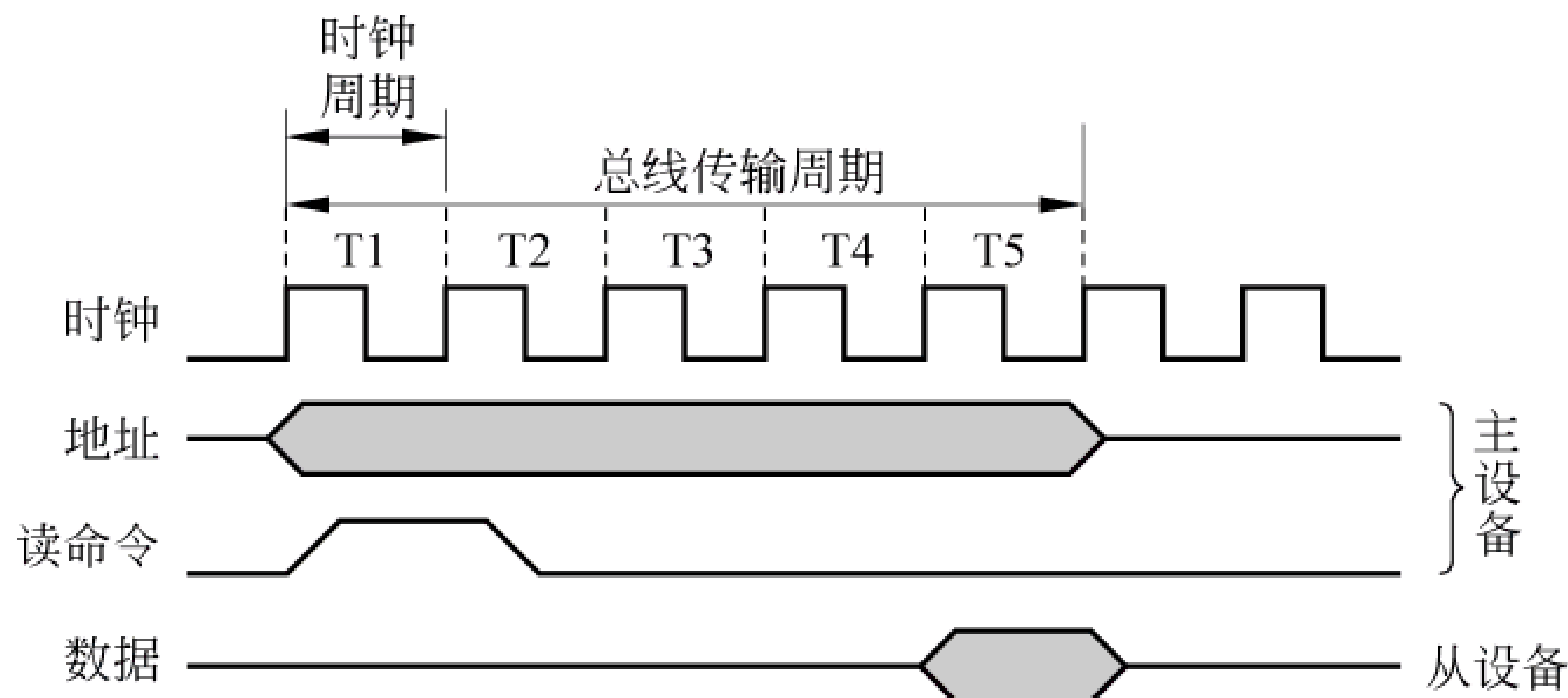


图8.4　同步通信协议（存储器读操作）

同步通信协议很容易用一个有限状态机实现，因为同步通信协议中每一步操作都是预先确定的，只涉及到非常少的逻辑，所以这种总线速度非常快，并且接口逻辑少。处理器-主存总线都是同步的，因为通信的设备靠得很近，而且数量又少。

同步通信方式有两个缺点。第一，总线定时以最慢设备所花时间为标准，所以同步总线适合于存取时间相差不大的多个功能部件之间的通信；第二，由于时钟偏移问题，导致同步总线不能过长，否则将会降低总线传输效率。此外，由于同步总线是由时钟信号来定时的，每一步操作的开始都不管前面操作结果是否正常结束，所以，可靠性不高。

2. 异步通信方式

异步总线不采用时钟定时，而是采用"异步应答"方式进行定时，异步通信协议也被称为"握手协议"。握手协议由一系列步骤组成，只有当双方都同意时，发送者或接收者才会进入

到下一步,协议是通过一组附加的“握手”控制信号线来实现的。因此,一个异步总线能够连接带宽范围很大的各种设备,总线能够加长而不用担心时钟偏移问题。

下面用一个简单的例子来说明异步总线如何工作。假定一个设备要求从存储器中读一个字,采用异步通信方式,总线中需要有以下三个控制信号线。

(1) ReadReq(读请求)信号。指示一个读请求开始,假定在送出该信号时,地址同时被放到地址线上。

(2) Ready(数据就绪)信号。指示数据已在数据线上准备好。在一个存储器读事务中,该信号由存储器驱动有效;在一个 I/O 读事务中,该信号由 I/O 设备驱动有效。

(3) Ack(回答)信号。用于回答另一方送过来的 ReadReq 或 Ready 信号。

图 8.5 给出了存储器读操作的异步通信协议示意图。

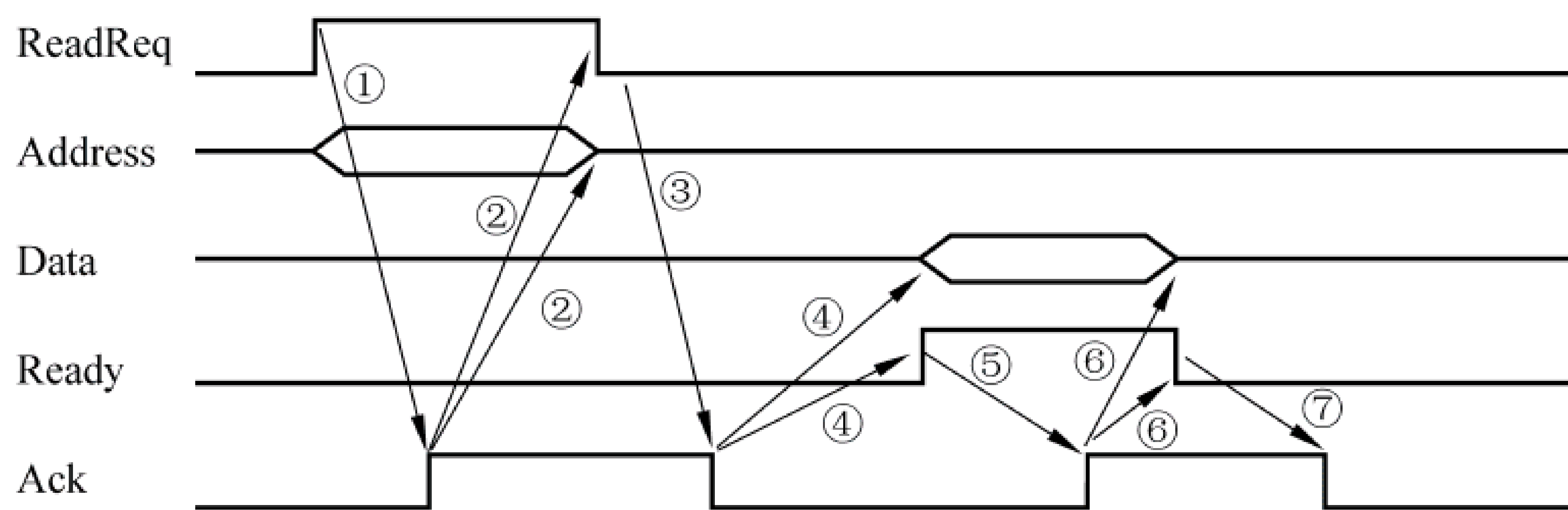


图 8.5　异步通信协议(存储器读操作)

一次存储器读总线操作从主设备送出一个读请求信号 ReadReq 和地址信息开始,共经历 7 个步骤。图中异步通信协议的每一步说明如下。

① 存储器接收到主设备送出的读请求 ReadReq 信号后,从地址线上读取地址信息,同时送出 Ack 信号,表示它已接受了读请求和地址信息。

② 主设备收到存储器送出的回答信号 Ack 后,接着就撤销读请求信号 ReadReq 和地址线。

③ 存储器发现读请求信号 ReadReq 被释放后,就跟着也撤销回答信号 Ack。

至此,一次完整的应答过程结束。在这个过程中完成了地址信号和读命令信号的交换。但一次总线操作还未完成。还要继续进行数据信息的交换。

④ 当存储器完成数据的读出后,就将数据放到数据线上,并送出数据就绪信号 Ready。

⑤ 主设备接收到存储器送出的数据就绪信号 Ready 后,就从数据线上开始读,并送出回答信号 Ack,向存储器表明数据已经被读。

⑥ 存储器接收到 Ack 信号后,得知数据已被成功读取。此时,它就撤销数据就绪信号 Ready,并撤销数据线。

⑦ 主设备发现 Ready 撤销后,跟着也撤销回答信号 Ack。

至此,又一次完整的应答过程完成。在这个过程中完成了一次数据信息的交换。

经历两次应答过程,一个存储器读事务就完成了。异步总线协议工作起来像一对有限状态机。它们以这样一种方式进行通信:一个设备直到它知道另一个设备已经到达一个特定的状态时才进入到下一状态,所以两个设备能协调工作。异步通信有非互锁、半互锁和全互锁三种可能的应答方式,如图 8.6 所示。

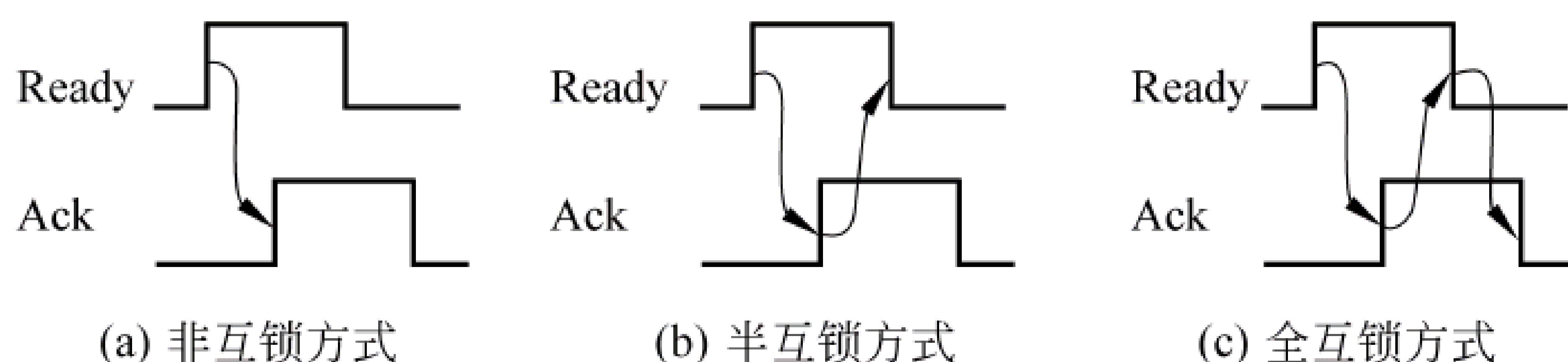


图 8.6 异步通信的三种互锁方式

• 非互锁方式

在非互锁方式中，发送部件将数据放在总线上，延迟一定时间后就发出数据就绪信号 Ready，通知对方数据已在总线上，接收部件根据这个就绪信号接收数据，并发出应答信号 Ack 作为回答，表示数据已接收到，发送部件收到 Ack 信号后就可撤销数据，以便进行下一次传输。这种方式下，只有一次握手，经过一个固定时间后，握手信号自动撤销。如果总线上各设备速度差异很大，就不能保证握手信号在规定时间内到达对方，因此这种非互锁方式在某些情况下不可靠。

• 半互锁方式

半互锁方式下，其握手过程类似于非互锁方式，只是让就绪信号 Ready 一直有效，直到接收到另一方送来的应答信号 Ack 为止。这种应答方式有两次握手，就绪信号的宽度由这两次握手信号控制，而回答信号在一个固定时间后自动撤销。

• 全互锁方式

在一个全互锁方式中，发送部件将数据放在总线上后就发出数据就绪信号 Ready，接收部件在接收到这个就绪信号后接收数据，并发出应答信号 Ack，发送部件收到 Ack 信号后才复位就绪信号，而接收部件在检测到就绪信号复位后，才复位应答信号。这种方式下，有三次握手，就绪信号和应答信号的宽度都由握手信号控制，都可依据传输情况变化，因而这种方式的异步总线可以挂接各种具有不同工作速度的设备。这种互锁式异步总线被广泛运用，上述图 8.5 中示意的就是这种全互锁方式，共有 7 次握手过程。

同步总线通常比异步总线快。因为异步通信需要进行握手，增加了开销。下面的例子将对同步和异步总线的性能进行简单的分析和比较。

例 8.2 假定同步总线的时钟周期为 50ns，总线上传输一个数据需要一个时钟周期，异步总线每次握手需要 40ns，两种总线的数据都是 32 位宽，存储器准备一个数据的时间为 200ns。要求求出从该存储器中读出一个字时两种总线的数据传输率。

解：同步总线的时钟周期为 50ns，通过同步总线进行存储器读所需的步骤和时间如下。

(1) 发送地址和读命令到存储器：50ns。

(2) 存储器读数据：200ns。

(3) 传送数据到设备：50ns。

因此，传送 32 位数的总时间为 300ns，故数据传输率为 4B/300ns＝13.3MBps。

直觉上可能会觉得异步总线慢得多，因为它将用 7 个步骤，每一步至少需 40ns，而且有关存储器访问那一步将用 200ns。但是，如果仔细看一下图 8.5，将会发现：其中有好几步是和主存访问时间重叠的，特别存储器在第①步结束接收到地址后，直到第⑤步的开始，才需要将数据放到数据总线上，因而，第②、③、④步都和存储器访问时间重叠。通过异步总线进行存储器读操作的步骤和时间为：

第①步为 40ns；第②、③、④步为 Max(3×40ns，200ns)=200ns；第⑤、⑥、⑦步为 3×40ns=120ns。因此，传送 32 位的总时间为 360ns，故数据传输率为 4B/360ns=11.1MBps。

由此可知，同步总线仅比异步总线快大约 20%。当然要获得这样的速度，异步总线上的设备和存储器系统必须足够快，以使每次在 40ns 内能完成握手过程。

尽管同步总线可能更快一些，但是，在同步方式和异步方式之间进行选择时，要考虑的不仅是数据传输速度，而且要考虑 I/O 系统的能力，包括可以连到总线上的设备的个数与总线的物理距离等。异步总线能更好地适应技术的改变，并能支持更大范围内的响应速度。

3. 半同步通信方式

异步通信采用异步应答方式，因而对噪声较敏感。为解决这个问题，一般在异步总线中引入时钟信号，规定握手信号总是在时钟信号的边沿被采样，这样，信号的有效时间仅限制在特定的时刻，而不受其他时间的信号的干扰。这种所有事件都由时钟信号定时、但信息的交换又由就绪和应答等握手信号控制的通信方式称为半同步通信方式。图 8.7 给出了一种半同步通信协议的示意图。

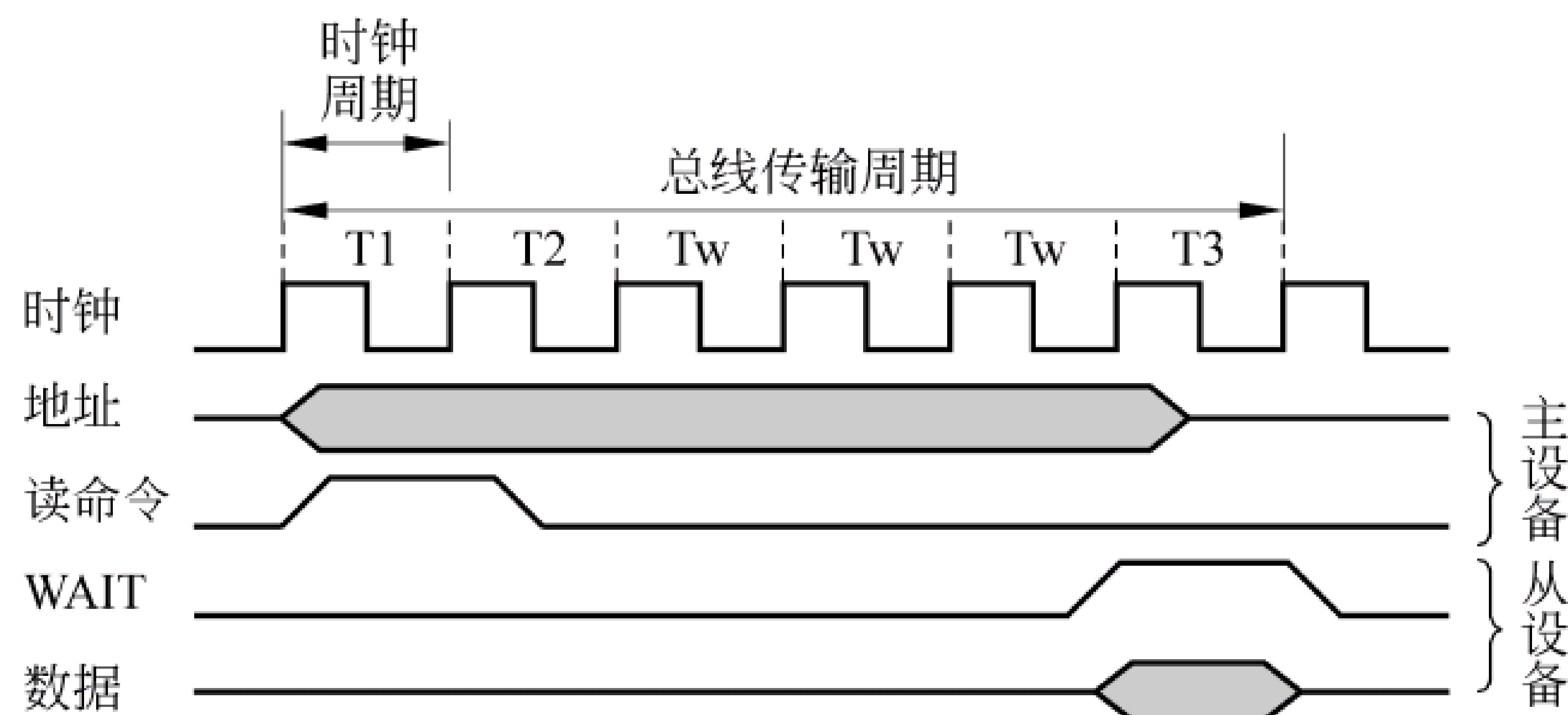


图 8.7　半同步通信协议

图 8.7 中，在同步时钟信号定时的基础上，引入一个 WAIT 信号，在时钟信号的上升沿，主设备采样 WAIT 信号，以察看从设备是否已完成相应任务。如果 WAIT 信号无效，则说明没有就绪，继续等待一个时钟，即插入等待时钟 Tw，直到 WAIT 信号有效为止。当采样 WAIT 信号有效时，主设备就开始从数据线上取数据。有些半同步总线对主设备和从设备各引入一条就绪信号线，如 PCI 总线中的“$\overline{\text{TRDY}}$”和“$\overline{\text{IRDY}}$”信号。

半同步通信同时具有同步和异步通信的优点，既保持了“所有信号都由时钟定时”的同步总线特点，又有“速度差异较大的设备可共存于同一总线”的异步总线特点。

***4. 分离事务通信方式**

在例 8.2 中，当存储器进行读操作时，若总线继续进行存储器读事务，那么总线将有 4 个时钟周期没有进行实际的传输。如果这时候释放总线，那么这 4 个时钟周期可被其他事务所用。这就是分离事务通信协议(Split Transaction Protocol)方式，可按如下步骤进行。

(1) 总线主控设备向存储器发出信号，送出地址和请求类型。

(2) 当存储器回答完请求后，主控设备释放所有信号线。

(3) 存储器访问开始，在存储器访问期间其他总线主控设备能使用总线。

(4) 存储器向主控设备发出信号表示数据已经可用。

（5）主控设备通过总线接收数据，并向存储器指示它已获取数据，存储器可从总线上撤销信号。

如图 8.8 所示，分离事务通信过程将一个传输操作事务的过程分成两个子过程。在第一个子过程中，主控设备 M 在获得总线使用权后，将请求的事务类型（即总线命令）、地址以及其他有关信息（如标识主控设备身份的编号等）发送到总线上，从设备 S 记录下这些信息。主控设备发送这些信息只需很短的时间，发完这些信息后便立即释放总线，这样其他设备便可使用总线。同时，从设备 S 收到主控设备 M 发来的信息后，按照其请求的命令进行相应的操作。在第二个子过程中，当从设备 S 准备好主控设备所需的数据后，便请求使用总线，一旦获得使用权，则从设备 S 就将主控设备 M 的编号及所需的数据等送到总线上，这样主控设备 M 便可接收数据。

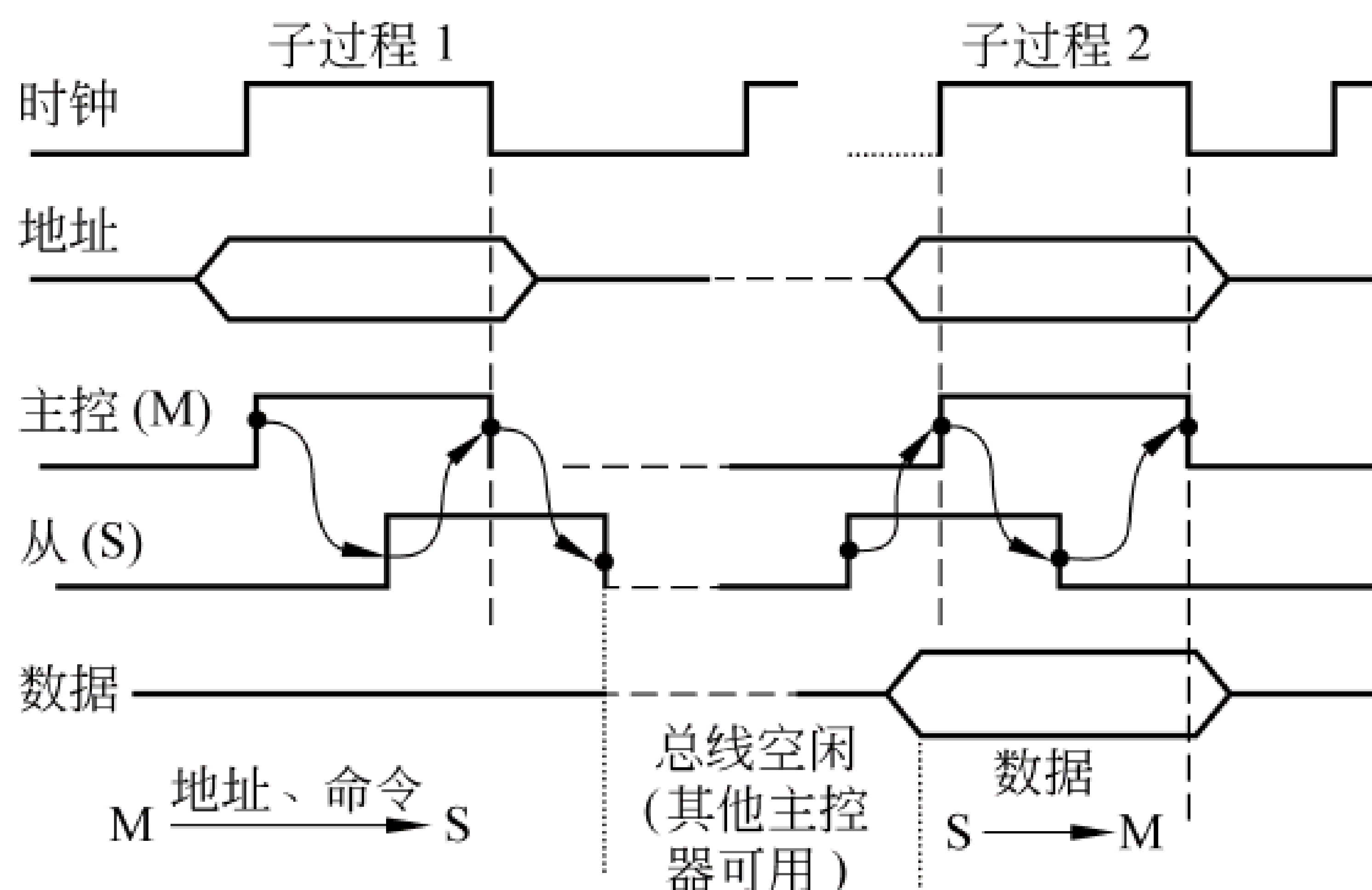


图 8.8　分离事务通信协议

分离通信方式的优点是在不传送数据期间释放总线，使其他申请者能使用总线，实现了一个总线为多个主/从设备进行交叉并行传送的方式，因而，可改进整个系统的总有效带宽。如果从设备准备数据的过程相当复杂，所需时间长，从而引起多个事务重叠时，效果更明显。当然这种方式控制相当复杂，一般在大型计算机系统和高档微机系统中使用。

事务的分离使得完成一个事务的时间可能会增加，因为必须要两次获得总线。分离事务协议实现起来开销大，主要是因为需要跟踪在一次通信中的另一方，在一个分离的事务协议中，从设备必须和请求方联系，以便在它完成读写操作后能告诉请求方，所以请求者的身份必须被传送并被从设备保存。

综上所述，总线在某种总线事务下的数据传输率与以下诸多因素有关。

（1）与总线带宽有关，总线带宽反映总线在进行数据传送时的传输速度，因此，它决定了一个总线事务中数据传送阶段所用的时间。总线带宽由总线宽度、时钟频率和传输每个数据所用时钟周期数确定，因此，总线事务的数据传输率与这些因素都有关。

（2）与总线裁决方式有关，总线裁决方式和裁决速度会影响总线事务所用时间。为了减少总线裁决时间，有些总线在数据传输时进行总线裁决，因而裁决阶段是隐含的，不占用总线事务时间。

（3）与总线定时方式有关，从上述例 8.2 可以看出，由于同步总线用时钟信号同步，而不像异步总线那样需要握手应答，因而比异步方式快。对于分离事务方式，由于实现了多个事务交叉并行传送，因而提高了总线的利用率，从而提高了总线的总体数据传输率。

(4) 与信号线类型有关,信号线分时复用有时会减慢总线的传输速度。例如,将地址线和数据线分时复用时,则在写操作的情况下不能将地址和数据同时传送出去,因而降低了数据传输率。

(5) 与是否采用突发传送(成组数据块传送)有关,突发传送方式下,只要传送一个首地址,然后就可连续传送多个数据而不释放总线,由于突发传送减少了数据传送过程中的地址传送和申请/释放总线等开销,因而提高了总线传输速率。

总线设计困难的一个原因是总线速度受到总线长度和部件个数的限制。有些技术可以用来提高总线的性能,但这些技术又可能反过来影响其他性能。例如,为了获得I/O操作的快速响应,可通过简化通信路径来使一次总线访问的时间降到最小;但另一方面,为了获得较高的I/O数据传输率,又必须使一次总线传输的数据量最大化,因此,缩短等待时间和提高传输速率这两个目标是相互矛盾的两种设计要求。此外,要求支持大范围内具有不同等待时间和不同数据传输率的设备的需求也使总线设计面临挑战。

*8.3 总线接口单元

总线上的信号必须与连到总线上的各个部件所产生的信号协调,起协调作用的控制逻辑就是总线接口,它是挂接在总线上的部件与总线之间的连接界面。总线接口单元是连接在总线上的部件中实现总线接口功能的那部分逻辑,因此,挂接在总线上的所有部件内部都有一部分逻辑是用于和总线进行连接的接口单元。CPU、存储器、I/O模块中都有相应的总线接口逻辑。

每个部件中的总线接口单元的功能可能不完全相同,这与部件本身的功能和结构、所连接的总线的特性等有关。但从大的方面来说,要考虑的基本问题是一样的。

总线接口单元的基本功能如下。

(1) 进行定时和通信。在同步通信方式下,提供或接收时钟信号,在时钟信号的控制下驱动或采样相应的信号线。在异步方式下,按照握手协议对相应的信号线进行驱动、复位(撤销)或采样。

(2) 总线请求和仲裁。根据需要发出总线请求信号。有些部件的总线接口中具有集中方式下的总线控制器,此时还要进行总线裁决。对于分布式裁决,每个总线接口都要参与裁决过程。

(3) 进行控制操作。提供命令译码等控制逻辑,以根据总线传送过来的命令启动总线部件进行相应的操作。

(4) 提供数据缓冲。在数据被送总线前缓存数据。当总线连接的部件之间有速度差异时,可以在接口中设置一些数据缓冲寄存器,利用这些寄存器使不同速度的部件得到匹配。

(5) 进行数据格式转换。当总线连接的部件之间数据格式不同时,可以通过接口进行数据格式转换。例如串-并转换、8位-32位转换等。

(6) 记录状态信息。外部设备的I/O总线接口还必须能够记录接口本身以及它所挂接的设备的状态。例如接口中数据缓冲的使用情况等。

(7) 数据传送控制。有些接口还要对数据传送过程进行控制,例如,对传输过程中的字计数器进行更新。

(8) 中断请求和响应。根据需要发出中断请求信号或接收中断请求并给出响应信号。例如,在外设的总线接口中,当外设需要向处理器请求某种服务时,它通过总线接口向CPU发中断请求信号。而处理器的总线接口则接收中断请求信号,并给出中断回答信号。

8.4 总线标准

主板上的“处理器-主存”总线经常是特定的专用总线,而用于连接各种I/O模块的I/O总线和底板式总线则通常可在不同的计算机中互用,因此,底板式总线和I/O总线通常是标准总线,可被不同公司制造的各种计算机系统所使用。

大多数计算机允许用户方便地插入设备,有的甚至是新开发的外围设备。I/O总线是在机器上扩充和连接新外设的一种常用手段。为了使机器的扩充和新设备的连接更加方便,计算机工业界已经开发出了各种总线标准,这些标准为计算机制造商和外围设备制造商提供了一种规范,只要新机器按照规范去设计,外设就能与其直接连接使用;与此同时,这些标准也向外设制造商保证,只要新外设按照规范去实现,用户的计算机就能挂接其新设备。有了总线标准,不同厂商可以按照同样的标准和规范生产各种不同功能的芯片、模块和整机,用户可以根据功能需求去选择不同厂家生产的、基于同种总线标准的模块和设备,甚至可以按照标准自行设计功能特殊的专用模块和设备,以组成自己所需的应用系统。这样可使芯片级、板卡级和设备级等各级别的产品都具有兼容性和互换性,使整个计算机系统的可维护性和可扩充性得到充分保证。

总线标准的形成有多种途径。第一种途径是由于流行而自然形成的标准。有些机器如此流行以至于它们的I/O总线最终变成了事实上的标准,例如IBM PC/AT总线。一旦一个总线标准被设备制造商大量使用,那么其他设备制造商也会引入这种总线并提供大量支持这种总线的设备。第二个途径是为了解决共性问题而提出一种标准。这种情况下,标准往往会由一个小组来制定。SCSI总线和Ethernet就是由多个制造商合作提出的标准总线的例子。第三种途径是通过标准化组织制定的。像ANSI或IEEE等组织会提出一些总线标准。PCI总线标准就是由Intel发起、后来由一个工业委员会发展起来的。

现在的标准总线规范越来越复杂,很多细节难以用很小的篇幅说明清楚。各种标准总线之间尽管在设计细节上有很多不同,并且各有特点,但从总体上看,无论哪种总线,在其规范中都包含了信号系统、电气特性和机械物理特性等。而信号系统的规定中通常又包含信号分类、数据宽度、地址空间、传输速率、总线仲裁、握手协议、总线定时和事务类型等内容。本节只对有代表性的PCI总线作较为详细的介绍,而其他I/O总线仅作简单介绍,有关细节请参阅相关资料。以下所列的总线标准中,虽然有些已经被淘汰,但为了便于对总线概念和总线标准有全面理解,本教材也将它们罗列出来。

*8.4.1 ISA总线

ISA(Industrial Standard Architecture)总线是IBM公司1984年为推出PC/AT而建立的系统总线标准,所以也叫AT总线。它是在原先的PC/XT总线的基础上扩充而来的。它在推出后得到广大计算机同行的承认,兼容该标准的微型机大量出现。并且随后出现的

286、386 和 486 微机尽管工作频率各异，但大多采用了 ISA 总线。

ISA 总线的主要特点如下。(1)它能支持 64K I/O 地址空间、16M 主存地址空间的寻址，可进行 8 位或 16 位数据访问，支持 15 级硬中断、7 级 DMA 通道。(2)它是一种简单的多主控总线。除了 CPU 外，DMA 控制器、DRAM 刷新控制器和带处理器的智能接口控制卡都可成为总线主控设备。(3)它支持 8 种总线事务类型：存储器读、存储器写、I/O 读、I/O 写、中断响应、DMA 响应、存储器刷新和总线仲裁。

它使用独立于 CPU 的总线时钟，因此 CPU 可以采用比总线时钟频率更高的时钟。它的时钟频率为 8MHz，共有 98 根信号线，在原 PC/XT 总线的 62 根线的基础上扩充了 36 根线，与原 PC/XT 总线完全兼容。它具有分立的数据线和地址线，数据线宽度为 16 位，可以进行 8 位或 16 位数据的传送，最大数据传输率(即总线带宽)为 16MBps。提供了两组地址信号线，即 SA19～SA0 和 LA23～LA17，使用这两组地址信号线可以对 16M 的主存地址空间和 64K 的 I/O 地址空间进行访问。

*8.4.2 EISA 总线

EISA(Extended Industrial Standard Architecture)总线是一种在 ISA 总线基础上扩充的开放总线标准。它从 CPU 中分离出了总线控制权，是一种具有智能化的总线，支持多总线主控和突发传输方式。它的时钟频率为 8.33MHz。EISA 总线共有 198 根信号线，在原 ISA 总线的 98 根线的基础上扩充了 100 根线，与原 ISA 总线完全兼容。具有分立的数据线和地址线。数据线宽度为 32 位，具有 8 位、16 位、32 位数据传输能力，总线带宽为 33MBps。地址线的宽度为 32 位，所以寻址能力达 2^{32}，即 CPU 或 DMA 控制器等这些主控设备能够对 4G 范围的主存地址空间进行访问。

*8.4.3 PCI 总线

Intel 公司联合 IBM、DEC、Apple、Compaq、Motorola 等 100 多家 PC 工业界的主要厂家，于 1992 年成立了 PCI 集团，统筹、强化和推广 PCI 标准。PCI 规范是公开的，它受到许多微处理器和外围设备生产商的支持，因此不同厂家生产的 PCI 产品是相互兼容的。

PCI(Peripheral Component Interconnect)是一种高带宽、独立于处理器的总线。它主要用于高速外设和主机相连，如图形显示适配器、网络接口控制卡、磁盘控制器等。它与 CPU 的时钟频率无关，采用 33MHz 或 66MHz 的总线频率，数据线宽度为 32 位，可扩充到 64 位，因此，总线带宽可达 133～533MBps。它支持无限突发传输，并支持并发工作，即挂接在 PCI 总线上的外设能与 CPU 并发工作。PCI 总线作为 CPU 和外设之间的一个中间层，一个或多个 PCI 总线通过 PCI 桥(PCI 控制器)和处理器总线相连，而处理器总线只连接处理器/cache、主存储器和 PCI 桥。PCI 桥的使用使 PCI 总线独立于处理器，并且 PCI 桥提供了数据缓冲功能。当处理器要访问 PCI 总线上的外设时，它可以把一批数据快速写到 PCI 桥的数据缓冲器中，在这些数据通过 PCI 总线写入设备的过程中，处理器又可以去执行其他操作。这种并发工作方式提高了系统的整体性能。

一个系统中可有多个 PCI 总线。图 8.9(a)和图 8.9(b)分别给出了在一个单处理器系统和多处理器系统中使用 PCI 总线的典型例子。



(a) 典型的单处理器系统



(b) 典型的多处理器系统

图 8.9　PCI 总线的典型配置

下面对 PCI 总线的信号线、PCI 命令、数据传送过程和总线裁决等内容做简要介绍。

(1) 信号线

PCI 可配置为 32 位或 64 位总线。表 8.2 给出了必需的 50 根信号线，按照功能它们被分为以下几组。

系统信号：包括时钟和复位线。

地址和数据信号：包含 32 根分时复用的地址/数据线、4 根分时复用的总线命令/字节允许线以及对这 36 根信号线进行奇偶校验的一根校验信号线。

接口控制信号：对总线事务进行定时控制，用于在事务的发起者和响应者之间进行协调。

裁决信号：它不同于其他信号，不是所有设备共享同一根信号线，而是每个总线主控设备都有一对仲裁线——总线请求和总线允许。PCI 采用集中式裁决，所有设备的仲裁线都连接到一个总线裁决器中。

错误报告信号：用于报告奇偶校验错以及其他错误。

表 8.2　PCI 必需的 50 根信号线

名　称	类型(根数)	说 明 描 述
系统信号		
CLK	in(1)	时钟信号,在其上升沿每个设备对相应的输入信号进行采样
$\overline{\text{RST}}$	in(1)	复位信号,使总线上的 PCI 专用寄存器、定序器和信号转为初始状态
地址和数据信号		
A/D[31∷0]	t/s(32)	复用的地址和数据线,地址阶段表示地址;数据阶段表示数据
C/BE[3∷0]	t/s(4)	复用的总线命令线和字节允许线。地址阶段表示总线命令;数据阶段表示数据线上 4 个字节中对应的那个字节是否有效
PAR	t/s(1)	32 根 A/D 线和 4 根 C/BE 线的偶校验信号线。地址阶段和写数据阶段由主设备驱动 PAR 信号线;在读数据阶段则由目标设备驱动 PAR 信号线
接口控制信号		
$\overline{\text{FRAME}}$	s/t/s(1)	由主设备驱动,表示总线传输已开始并在持续进行中。在总线传输的开始(即地址阶段之初)使该信号有效,而在进行总线传输的最后一个数据交换之前撤销该信号
$\overline{\text{IRDY}}$	s/t/s(1)	发送端就绪信号,由主设备驱动。读操作中表示主设备准备好接受数据;写操作中表示主设备已把有效数据放到 A/D 线上
$\overline{\text{TRDY}}$	s/t/s(1)	接受端就绪信号,由从设备驱动。写操作中表示从设备准备好接受数据;读操作中表示从设备已把有效数据放到 A/D 线上
$\overline{\text{STOP}}$	s/t/s(1)	由从设备驱动,表示希望主设备停止当前的总线传输操作
$\overline{\text{LOCK}}$	s/t/s(1)	表示正在进行的总线操作不可被打断,即锁定总线
IDSEL	in(1)	设备选择初始化信号,在配置读和配置写事务中用作片选信号
$\overline{\text{DELSEL}}$	in(1)	设备选择信号,如果某个目标识别出地址线上给定的是自己的地址的话,那么该设备就使这根线有效。主控设备接受到该信号后就知道已有设备被选中
仲裁线		
$\overline{\text{REQ}}$	t/s(1)	总线请求线,由需要申请总线使用权的主控设备发出。这是一根与设备有关的点对点信号线
$\overline{\text{GNT}}$	t/s(1)	总线允许线,接受到该信号的设备将获得总线使用权。这也是一根与设备有关的点对点信号线
错误报告信号		
$\overline{\text{PERR}}$	s/t/s(1)	表示一个目标在写数据阶段或一个主控设备在读数据阶段检测到一个奇偶校验错
$\overline{\text{SERR}}$	0/d(1)	可由任何一个设备发出。用以报告地址校验错或除校验错以外的其他严重错误

PCI 规范还定义了另外 50 根可选的信号线,它们分为以下几个功能组。

中断信号:同裁决信号一样也是非共享信号。每个 PCI 设备有自己的中断请求线,被连到中断控制器。

cache 支持信号:这些信号用来对 PCI 总线上的存储器提供 cache 支持。如果系统允

许 PCI 总线上的存储器支持 cache 的话，那么，当访问 PCI 总线上的某个存储单元时，必须提供相应的 cache 支持信号，用于传递 cache 侦听的结果。

64 位总线扩展信号：包含 32 根分时复用的地址/数据线、4 根分时复用的总线命令/字节允许线以及对这 36 根信号线进行奇偶校验的一根校验信号线。它们与基本的 32 位地址和数据信号组合形成 64 位地址和数据信号。另外，还有一对要求进行 64 位传输的请求和回答信号线。

JTAG/边界扫描信号：用于支持 IEEE 标准 194.1 中定义的测试程序。

(2) PCI 命令

总线活动以发生在总线主控设备和从设备之间的总线事务形式进行。总线主控设备是事务的发起者，从设备是事务的响应者，即目标。当总线主控设备获得总线使用权后，在总线事务的地址周期，通过分时复用的总线命令/字节允许信号线 C/BE 发出总线命令，以指出事务类型。

PCI 支持的总线命令(事务类型)有以下几类。

中断响应：它是一条读取中断向量的命令。用于对 PCI 总线上的中断控制器提出的中断请求进行响应。在该事务的地址周期地址线不起作用。而在数据周期，则从中断控制器读取一个中断向量，此时 C/BE 信号线用于表示读取的中断向量的长度。

特殊周期：用于一个总线主控设备向一个或多个目标广播一条消息。

I/O 读和 I/O 写：用于在事务发起者和一个 I/O 控制器之间进行数据传送。

存储器读、存储器行读、存储器多行读：用于总线主控设备从存储器中读取数据。PCI 支持突发传送，所以它将占用一个或多个数据周期。这些命令的解释依赖于总线上的存储控制器是否支持 PCI 的高速缓存协议。如果支持的话，那么，与存储器之间的数据传送以 cache 行的方式进行。三条存储器读命令的含义见表 8.3。

表 8.3　PCI 总线上存储器读命令的含义

读命令类型	支持 cache 的内存	不支持 cache 的内存
存储器读	突发传送半个或不到一个 cache 行	突发传送两个数据周期或更少
存储器行读	突发传送半个以上到三个 cache 行	突发传送三个到 12 个数据周期
存储器多行读	突发传送三个以上 cache 行	突发传送 12 个以上数据周期

存储器写、存储器写并无效：这两种存储器写命令用于总线主控设备向存储器写数据，它们将占用一个或多个数据周期。其中存储器写并无效命令用于回写一个 cache 行到存储器，因此，它必须保证至少有一个 cache 行被写回。

配置读、配置写：用于一个总线主控设备对连接到 PCI 总线上的设备中的配置参数进行读或更新。每个 PCI 设备都有一个寄存器组(最多可有 256 个寄存器)，这个寄存器用于系统初始化时对本设备进行配置。

双地址周期：由一个事务发起者用来表明它将使用 64 位地址来寻址。

(3) 数据传送过程

PCI 总线上的数据传送由一个地址周期和一个或多个数据周期组成。图 8.10 显示了

一个读操作的时序。写操作的过程与读操作类似。

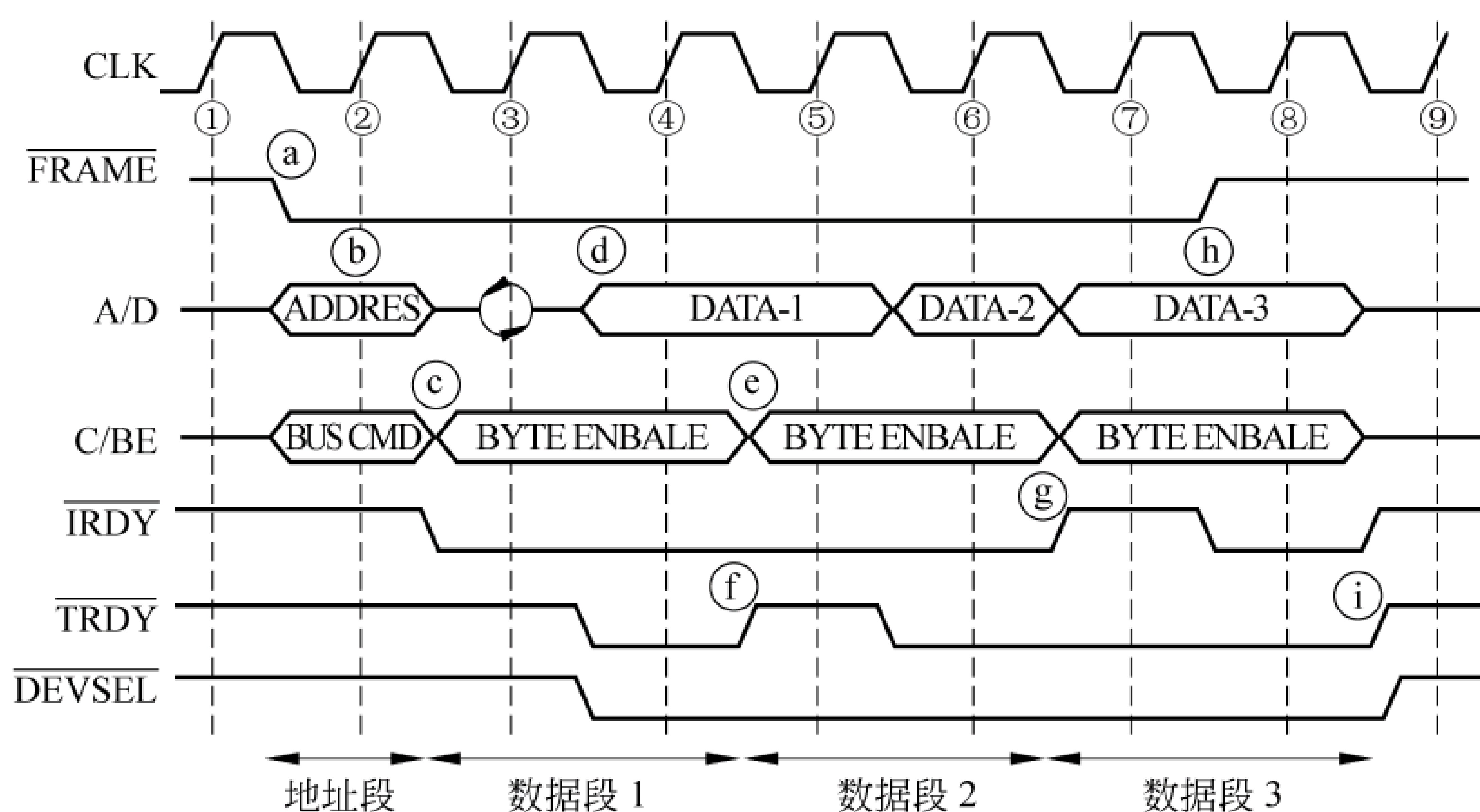


图 8.10　PCI 读操作过程

PCI 总线事务中，所有事件在时钟下降沿（即时钟周期的中间）同步。总线设备在一个时钟周期开始的上升沿采样总线信号。下面对图 8.10 中标出的一些重要事件进行说明。

① 总线主设备一旦获取总线使用权，就通过使$\overline{\text{FRAME}}$信号有效来开始一个事务，同时事务发起者还将起始地址送到 A/D 线上，将“读”命令送到 C/BE 线上。$\overline{\text{FRAME}}$信号线一直有效，直到它准备传送最后一个数据。

② 在第二个时钟的开始处，目标设备识别出 A/D 线上的地址。

③ 事务发起者停止驱动 A/D 线上的信号。所有可能由多个设备驱动的信号线都要求有一个换向周期（用两个环形箭头表示）。这样，地址信号的撤销将使 A/D 信号线下次为目标设备所用。同时，事务发起者将改变 C/BE 线上的信号，以指示 A/D 线上传送的有效数据是哪些字节。此外事务发起者还使$\overline{\text{IRDY}}$信号有效，表明它已准备好接收第一个数据。

④ 被选中的目标设备使$\overline{\text{DEVSEL}}$信号有效，以表明它已经识别出地址，并即将响应事务。它将请求的数据放到 A/D 线上，然后使$\overline{\text{TRDY}}$信号有效，表明 A/D 上的数据已经有效，可以读取。

⑤ 在第 4 个时钟开始处，事务发起者读取数据，并改变 C/BE 线上的字节允许信号，为下一次读做准备。

⑥ 在图示例子中，假定目标设备需要一些时间来准备第二个数据，所以它将$\overline{\text{TRDY}}$信号撤销，以通知事务发起者下一周期没有新数据。这样，事务发起者在第 5 个周期开始时就不会读数据，并且在这个周期中不改变 C/BE 线上的字节允许信号。第二个数据在第 6 时钟开始时被读取。

⑦ 在第 6 周期中间，目标设备将第 3 个数据放到总线上。在这个例子中，假定事务发起者没有准备好接收数据（例如，用来存放数据的缓冲已满无法接受新数据），所以，它将取消$\overline{\text{IRDY}}$信号，这样就会使目标设备将第 3 个数据继续在 A/D 线上多保持一个时钟周期。

⑧ 事务发起者知道第 3 个数据是最后一个，所以它就将$\overline{\text{FRAME}}$信号撤销，以通知目标设备这是最后一次数据传送。同时，它将$\overline{\text{IRDY}}$信号有效，表示它已准备好接收数据。

⑨ 事务发起者接受完数据，撤销$\overline{\text{IRDY}}$信号，使总线变为空闲。同时目标设备也相应地

撤销$\overline{\mathrm{TRDY}}$和$\overline{\mathrm{DEVSEL}}$信号。

（4）总线裁决

PCI 采用独立请求集中裁决方式。每个总线主设备都有两个独立的请求线$\overline{\mathrm{REQ}}$和允许线$\overline{\mathrm{GNT}}$。PCI 总线规范没有规定具体的仲裁算法。总线仲裁器可以使用静态的固定优先级法、循环优先级法或先来先服务法等仲裁算法。PCI 必须为它的每个总线事务进行仲裁，而且可以在总线进行数据传送的时候进行仲裁，因此仲裁不会浪费总线周期。这种方式称为隐式仲裁。

图 8.11 给出设备 A 和设备 B 在 PCI 上进行总线仲裁的例子。

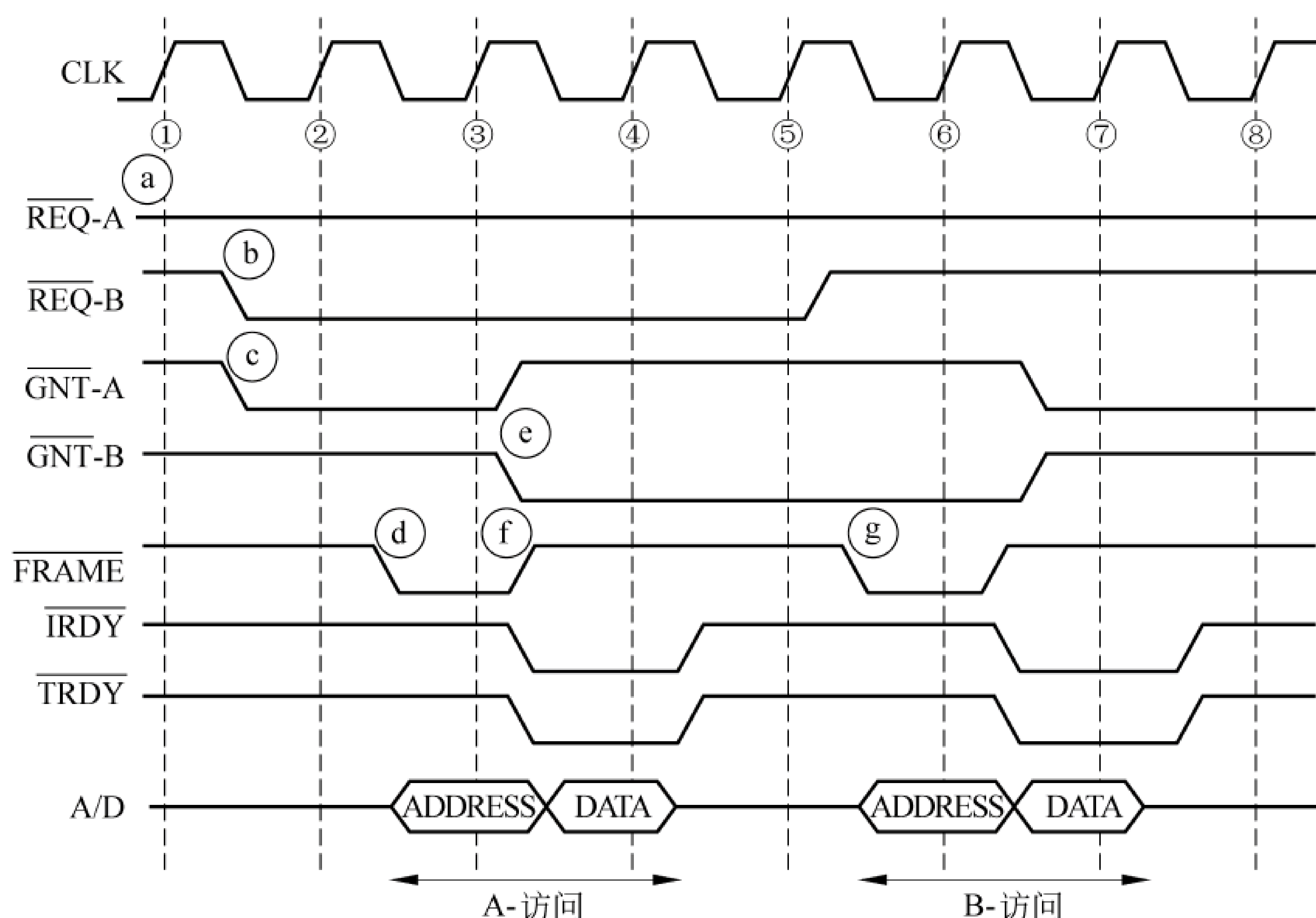


图 8.11　PCI 总线裁决过程

图 8.11 中设备 A 和设备 B 的总线仲裁过程说明如下。

① 在第 1 个时钟开始前，设备 A 已经使它的$\overline{\mathrm{REQ}}$信号有效，总线裁决器在第一个时钟开始时检测到这一信号。

② 在第 1 个时钟周期中，设备 B 使它的$\overline{\mathrm{REQ}}$信号有效，也请求使用总线。

③ 同时，总线裁决器向设备 A 发出$\overline{\mathrm{GNT}}$信号，授权设备 A 可以访问总线。

④ 设备 A 在第二个时钟开始检测到$\overline{\mathrm{GNT}}$信号，知道自己已被允许使用总线。同时，它发现$\overline{\mathrm{IRDY}}$和$\overline{\mathrm{TRDY}}$信号都没有声明，表明总线空闲。因此，它将$\overline{\mathrm{FRAME}}$信号有效，表明它要使用总线，同时将地址和命令信息放到相应的信号线上。因为设备 A 希望继续进行第二次交换，所以继续让 REQ 有效。

⑤ 总线裁决器在第 3 个时钟开始采样所有$\overline{\mathrm{REQ}}$线，根据裁决算法决定由设备 B 进行下一次总线事务。这时，它向设备 B 发回$\overline{\mathrm{GNT}}$信号，而撤销设备 A 的$\overline{\mathrm{GNT}}$信号。但设备 B 只有等到总线空闲后才能使用总线。

⑥ 设备 A 撤销$\overline{\mathrm{FRAME}}$信号，表示正在进行最后一次数据传送。它将数据放到数据总线上，并用$\overline{\mathrm{IRDY}}$通知目标设备，目标设备在下一个周期读取数据。

⑦ 在第5个时钟开始，设备B发现$\overline{\text{IRDY}}$和$\overline{\text{TRDY}}$信号都没有声明，表明总线是空闲的。因此，它将$\overline{\text{FRAME}}$信号有效表明它要使用总线，它同时将地址和命令信息放到相应的信号线上。因为它不希望继续进行第二次交换事务，所以它撤销了$\overline{\text{REQ}}$信号。

在设备B完成它的事务后，设备A将获得总线使用权，以进行它的第二次数据交换事务。

从上述过程来看，PCI的总线仲裁确实和数据传送同时进行，因此，PCI的事务过程中只有一个地址周期和若干个数据周期组成，不包含裁决周期。

从上述介绍的第一个版本PCI 1.0(33MHz/32b)公布以来，已经发展出了多个版本：PCI 66MHz/32b、PCI 33MHz/64b、PCI 66MHz/64b，1999年又出现了更高带宽的PCI-X总线，并且有多个PCI-X版本。

8.5 总线结构

计算机系统中采用的总线结构有单总线结构和多总线结构。

*8.5.1 单总线结构

早期的计算机采用单总线结构方式。它将CPU、主存和I/O模块都挂接在一个总线上，CPU与主存、CPU与I/O模块、主存与I/O模块之间的传送都通过同一组总线进行，如图8.12所示。PDP-11和国产DJS183机采用这种单总线结构。

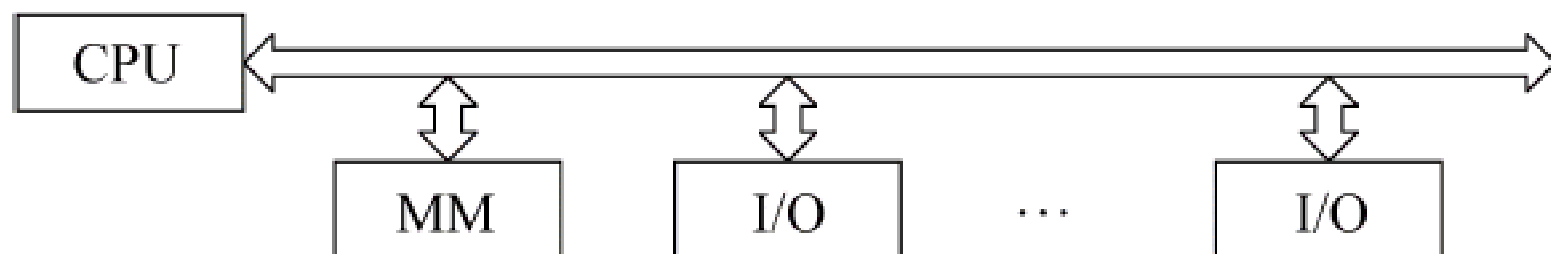


图8.12 单总线结构

这种总线结构简单、便于扩充，但所有传送都共享一组总线，极易使总线成为整个系统的瓶颈。因为一个总线上某一时刻只能有一对设备进行传输，所以，所有设备只能分时共享总线。随着计算机应用领域的扩大，挂接在系统中的外设种类和数量越来越多，对数据传输的速度要求也越来越高，如果还用这种单总线方式，性能就会急剧下降。因为总线上连接的设备越多，传输延迟就会越大。当控制信号频繁地从一个设备传递到另一个设备时，这种传输延迟就会明显地影响性能。在数据传输需求量和速度要求不高的情况下，可以通过增加总线宽度和提高总线的时钟频率来解决总线瓶颈问题。但当总线上挂接了大量的高速设备(如视频和图形控制器、LAN控制器等)后，单一总线就再也无法满足系统的要求了。因此，要既能为键盘、Modem等这些慢速设备传送数据，也能为高速设备传送数据，而且要把CPU从数据传送操作中解放出来，只有采用多总线结构，并根据数据传输的不同要求采用分层次的总线结构。

*8.5.2 双总线结构

在单总线的基础上再开辟一条CPU与主存之间的通路，形成以主存储器为中心的双

总线结构,如图 8.13(a)所示。CPU 与主存间的通路称为主存总线。该总线的速度较高,而且由于只在主存与 CPU 之间传输信息,因而速度快,效率高;同时,它又减轻了系统总线的负担,并且保证主存与 I/O 之间也能直接传送,而不需通过 CPU。国产 DJS184 机采用该总线结构。

图 8.13(b)也是一种双总线结构,它主要用在采用输入输出处理器(IOP)方式进行 I/O 传送的计算机系统中。其基本思想是将 I/O 设备从单总线分离出来,将原先的单总线分成主存总线和 I/O 总线。CPU、主存和输入输出处理器之间的信息传送在主存总线上进行;而各种 I/O 设备与主机之间的信息交换通过 I/O 总线和主存总线进行。输入输出处理器是一种专门用于进行输入输出控制的特殊处理器,它将 CPU 中大部分 I/O 控制任务接管过来,从而具有对各种 I/O 设备进行统一管理的功能。这种总线结构是一种分层的总线结构。通过 IOP 将 CPU 和 I/O 分离开来,减轻了 CPU 参与 I/O 而带来的负担。一般这种结构的系统会将不同特性的外设分类挂接在输入输出处理器的不同通道上。

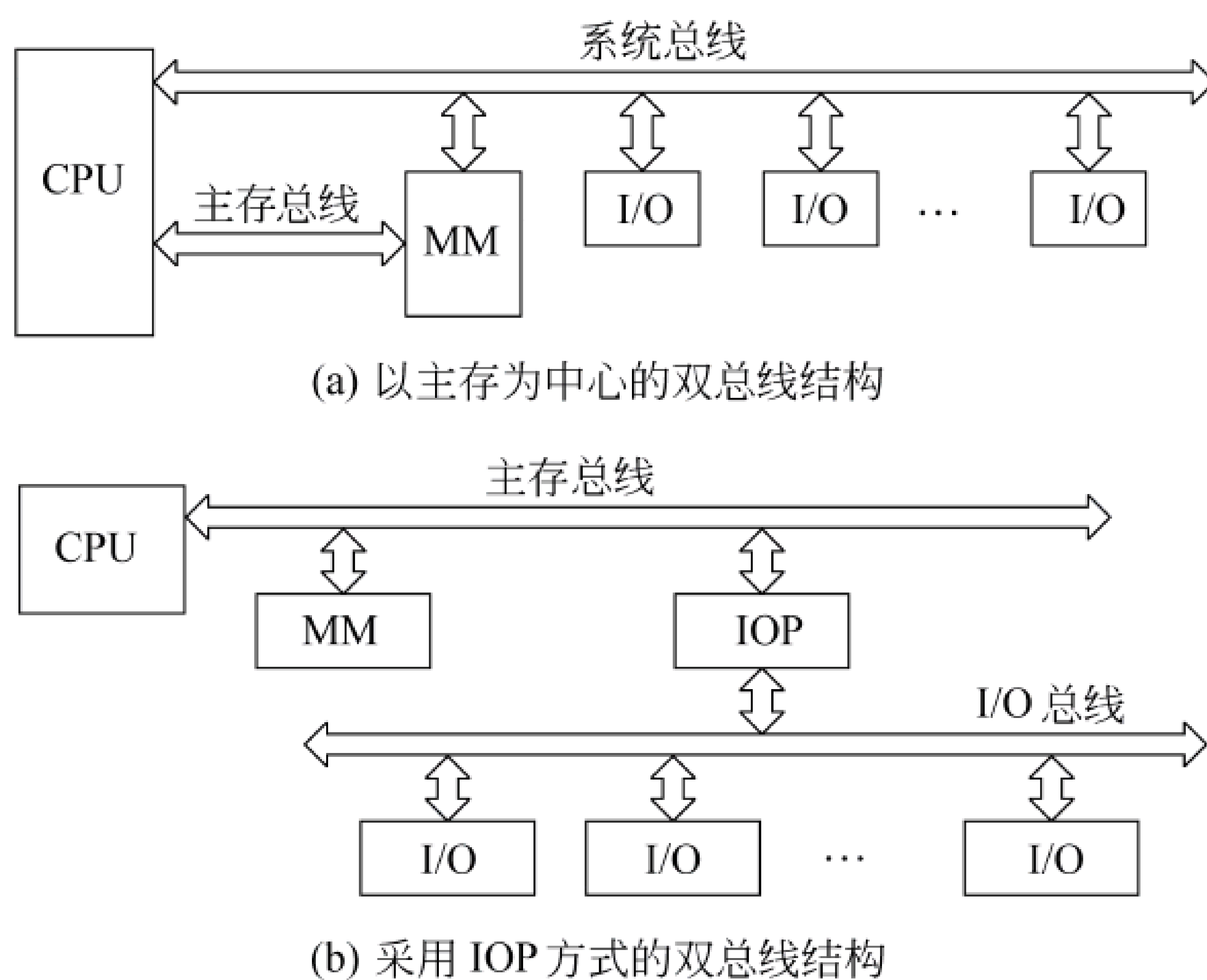


图 8.13　双总线结构

8.5.3　多总线结构

如果在上述以主存为中心的双总线结构中,将 I/O 设备和主存从系统总线上分离出来,将原先的系统总线分成主存总线和 I/O 总线。而在主存和高速的磁盘等设备之间引入一个专门的 DMA 总线,那么系统可构造一种三总线结构,如图 8.14 所示。在这种总线结构中,主存总线用于 CPU 和主存之间的信息传送,I/O 总线用于 CPU 和各个 I/O 之间进行信息传输,DMA 总线用于高速外设和主存之间的信息交换。在这种三总线结构中,DMA 总线和主存总线不能同时用于访问主存。

还有一种传统的总线结构采用处理器-cache 总线、主存总线、I/O 总线三级总线结构,如图 8.15 所示。处理器和高速缓存之间通过专门的局部总线相连,并且可将其他靠近 CPU 的局部设备连接到该总线。高速缓存同时还与主存储器一起连接到主存总线上,它们之间通过主存总线进行数据传输。这种结构引入了一条或多条扩展 I/O 总线(如 ISA、EISA、MCA 总线等),主存总线和扩展 I/O 总线上 I/O 设备之间的数据传送可以通过扩展

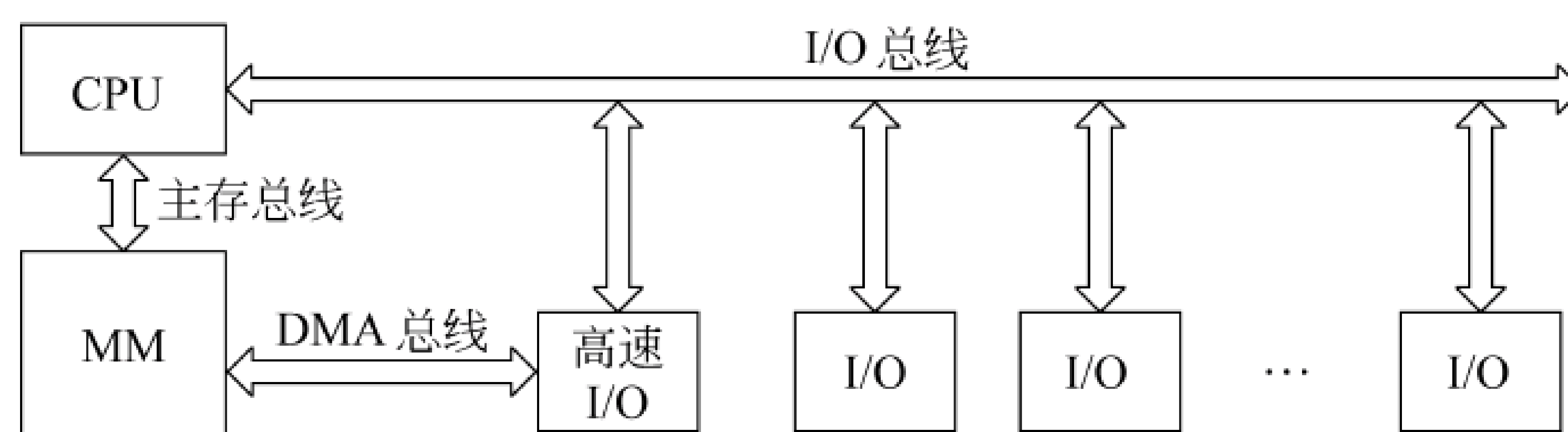


图 8.14　早期的三总线结构

总线接口来缓冲，这种设置使得系统能够支持更广泛的 I/O 设备，并且将 I/O 设备和主存间的通信与处理器的活动隔离开来。

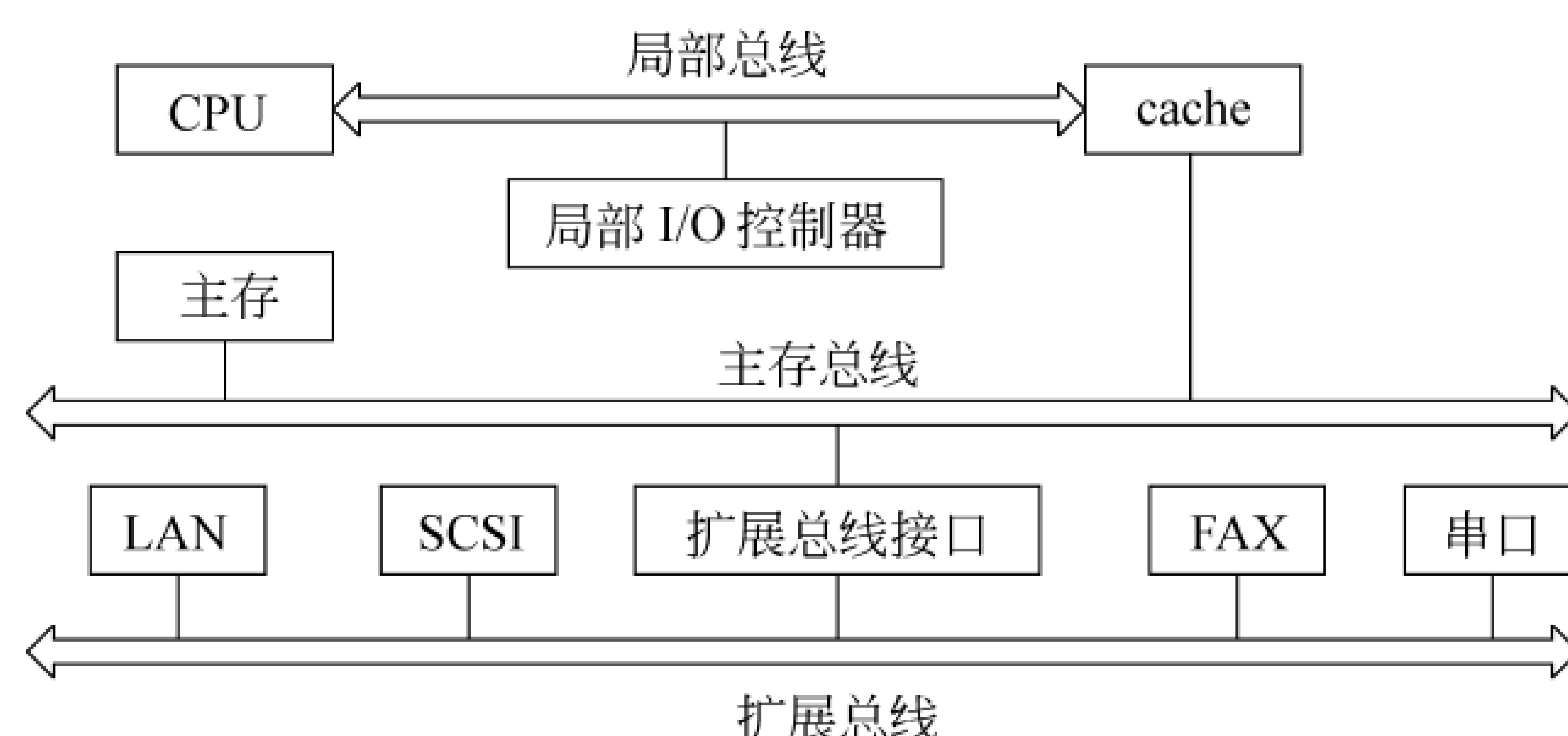


图 8.15　传统的三级总线结构

图 8.15 所示总线结构在 I/O 设备性能都相差不大的情况下比较有效，但随着计算机技术和应用水平的不断提高，cache 已经做在 CPU 中，并且大量高性能外设不断涌现，高速的视频图形设备、LAN 和 SCSI 设备等如果还和低速的串行接口、打印机等连在同一个总线上的话，那么势必会影响系统的效率。因此，典型的做法就是在主存总线和扩充 I/O 总线之间引入一种高速总线（如 VL 总线、PCI 总线等），将那些高速的块传送设备挂接在这种高速总线上。而低速 I/O 设备仍然由扩充 I/O 总线支持。如图 8.16 所示即是这种改进的多级总线结构示意图。

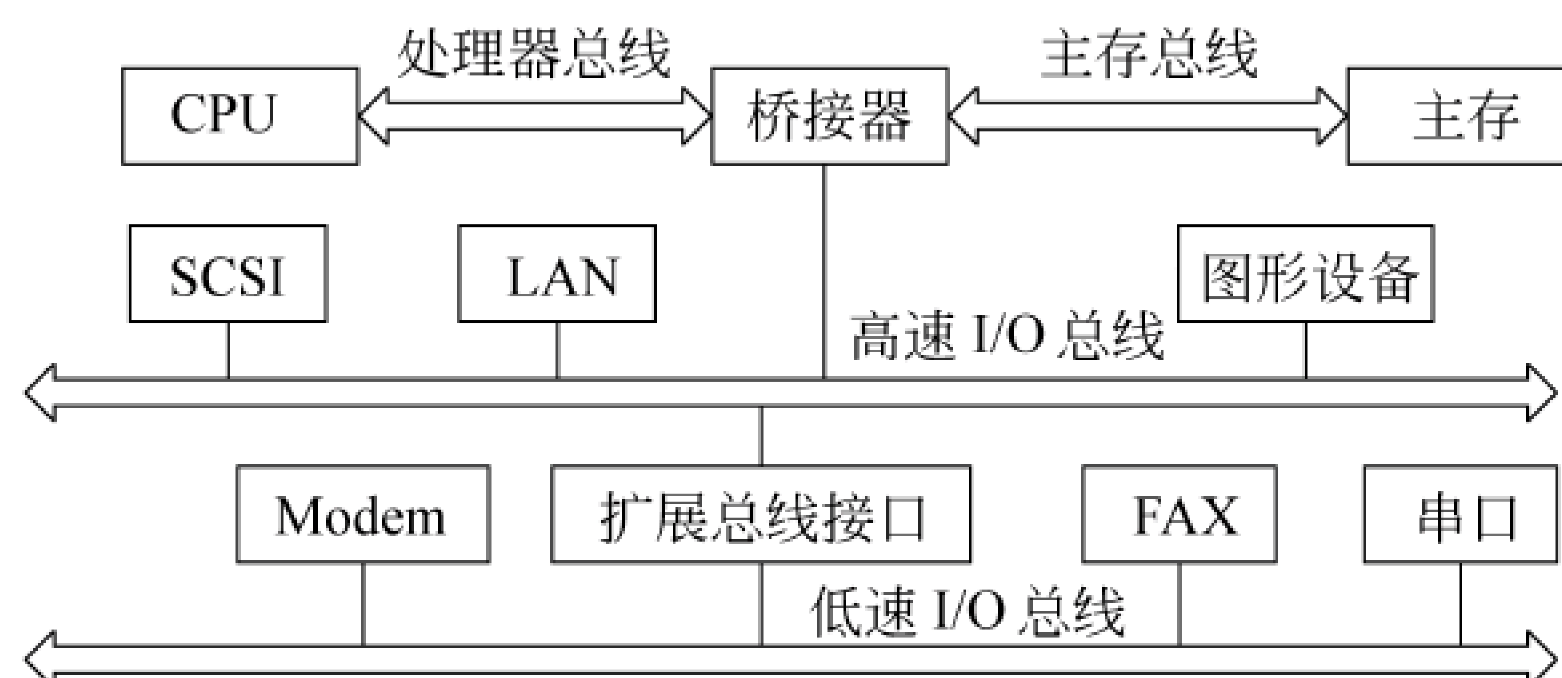


图 8.16　改进的多级总线结构

图 8.17 是一个典型的基于奔腾 4 处理器系统的总线结构。它反映了处理器总线、存储器总线、PCI 总线以及外设接口（AGP 接口、USB 接口、ATA 接口、串口 COM 及并口 LPT）连接 CPU、主存和各种外设的连接关系。从图中可以看出，越靠近 CPU 的总线越快，越远离 CPU 的总线越慢。

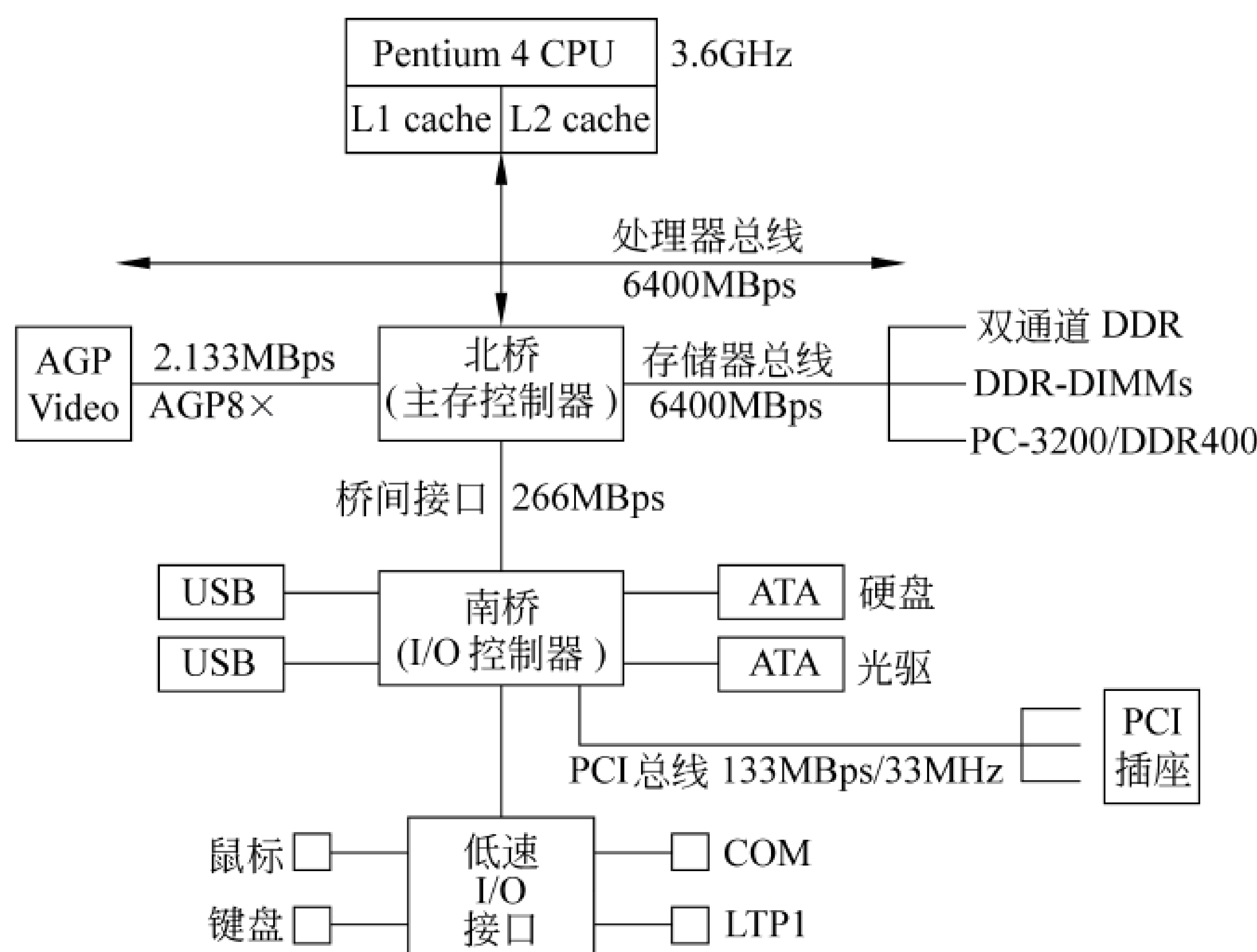


图 8.17　典型的 Pentium 4 系统总线结构

8.6　本章小结

本章介绍了系统总线的有关概念。主要包括总线的基本概念、总线的分类、总线的组成及性能指标、总线裁决、总线定时方式、总线标准和总线结构等。

具体总结如下。

- 总线的概念：总线是计算机中部件之间传送信息的公共通路，包括传输介质和相应的控制逻辑。
- 总线的分类
  - 内部总线：指芯片内部连接各元件的总线。
  - 系统总线：指在计算机的主要功能部件(CPU、主存和 I/O)之间传送信息的总线，由数据线、地址线和控制线组成。根据所处位置和功能的不同，系统总线有处理器总线、存储器总线和 I/O 总线。通常处理器总线和存储器总线是专用的主板式总线，而 I/O 总线是标准总线。
  - 通信总线：指用于主机和 I/O 设备之间或计算机系统之间通信的总线。
- 总线带宽：指总线的最大数据传输率，即在数据传输时单位时间内总线上可传输的数据量。它与总线位宽、总线时钟频率和传送每个数据所花的时钟周期数有关。
- 总线事务：指在总线上进行的不同信息传输类型，如存储器读、存储器写、I/O 读、I/O 写、中断响应等。有些总线事务要求完成一连串连续单元的读写，如从存储器读一个 cache 行或写一个 cache 行到主存，这种情况下，一个总线事务完成多个数据的读写，称为突发(burst)传输方式。
- 总线裁决方式
  - 集中裁决：有专门的总线裁决器确定总线使用权。

    ▲ 链式查询：总线允许信号在设备间用菊花链串联，按顺序查询，接口简单。
    ▲ 计时器定时查询：通过计数值确定查询顺序，优先级灵活，能保证公平性。
    ▲ 独立请求：每个设备有独立的总线请求线，优先级可编程设置，速度快。
  ◆ 分布式裁决：没有专门的总线裁决器，裁决逻辑分散在每个设备的总线接口中。
    ▲ 自举分布：各设备查看到所有优先级比自己高的设备没有请求时使用总线。
    ▲ 冲突检测：直接使用总线，使用过程中侦听到有其他设备也在使用总线时，则释放总线，然后在一个随机时间段后再次使用总线。
    ▲ 并行竞争：每个设备接口中有专门的仲裁逻辑，能保证送到数据线上的多个仲裁号（设备号）中只有最大的仲裁号的信息保留在数据线上。
- 总线定时方式
  ◆ 同步：用一个公共的时钟信号对传输过程的每个步骤进行同步控制。
  ◆ 异步：用异步应答信号对传输过程的每个步骤进行定时控制。
  ◆ 半同步：结合同步和异步两种定时方式，在时钟控制下发出和采样应答信号。
  ◆ 分离事务：把传输过程分成两个阶段，使得从设备在准备数据时总线被释放给其他设备使用。
- 总线标准：有 ISA、EISA、PCI 等。
- 总线结构
  ◆ 单总线结构：所有主要功能部件（CPU、主存和 I/O 模块）都挂接在一个总线上。
  ◆ 双总线结构：CPU、主存、I/O 之间分别互连，形成不同的总线。例如，CPU 和主存之间用处理器-主存总线，CPU 和 I/O 之间用 I/O 总线；CPU、主存和 IOP 之间用主存总线，各 I/O 和 IOP 之间用 I/O 总线。
  ◆ 多总线结构：将不同速度的部件细分后再分级互连，总线和总线之间用桥接器相连，以形成多总线结构。如 cache 和 CPU 之间单独用局部总线相连；处理器总线和存储器总线之间加一个桥接器，分别与处理器和存储器相连；高速 I/O 设备和低速 I/O 设备分离，分别用高速 I/O 总线和慢速 I/O 总线相连。

习　题　8

1. 给出以下概念的解释说明。

(1) 总线	(2) 片内总线	(3) 系统总线	(4) 通信总线
(5) 并行总线	(6) 串行总线	(7) 底板总线	(8) I/O 总线
(9) 处理器总线	(10) 存储器总线	(11) 信号线复用	(12) 总线传输周期
(13) 总线宽度	(14) 总线时钟频率	(15) 总线带宽	(16) 总线事务
(17) 主设备	(18) 从设备	(19) 突发传送	(20) 总线裁决
(21) 总线仲裁器	(22) 总线请求信号	(23) 总线允许信号	(24) 集中裁决方式
(25) 分布裁决方式	(26) 同步总线	(27) 异步总线	(28) 握手信号
(29) 半同步总线	(30) 分离事务协议		

2. 简单回答下列问题。

(1) 什么情况下需要总线仲裁? 有哪几种常用的仲裁方式? 各有什么特点?

(2) 总线通信采用的定时方式有哪几种? 各有什么优缺点?

(3) 在异步通信中,握手信号的作用是什么?

(4) 什么是突发传送方式?

(5) 提高同步总线的带宽有哪几种措施?

(6) 制定总线标准的好处是什么? 总线标准是如何制定出来的?

3. 假设一个同步总线的时钟频率为50MHz,总线宽度为32位,每个时钟周期传送一个数据,该总线的最大数据传输率(即总线带宽)为多少? 若要将该总线的带宽提高一倍,可以有哪几种方案?

4. VAX SBI总线采用分布式的自举裁决方案,总线上每个设备有唯一的优先级,而且有一根独立的总线请求线REQ,SBI有16根这样的请求线(REQ0,…,REQ15),其中REQ0优先级最高,请问,最多可有多少个设备连到这样的总线上? 为什么?

5. 假定一个32位微处理器的外部处理器总线的宽度为16位,总线时钟频率为40MHz,假定一个总线事务的最短时间是4个总线时钟周期,则该总线的最大数据传输率是多少? 如果将外部总线的数据线宽度扩展为32位,那么该总线的最大数据传输率提高到多少? 这种措施与加倍外部处理器总线时钟频率的措施相比,哪种更好?

6. 试设计一个采用固定优先级的具有4个输入的集中式独立请求裁决器。

7. 假设某存储器总线采用同步定时方式,时钟频率为50MHz,每个总线事务以突发方式传输8个字,以支持块长为8个字的cache行读和cache行写,每字4字节。对于读操作,访问顺序是一个时钟周期接受地址,三个时钟周期等待存储器读数,8个时钟周期用于传输8个字。对于写操作,访问顺序是一个时钟周期接受地址,两个时钟周期延迟等待,8个时钟周期用于传输8个字,三个时钟周期恢复和写入纠错码。对于以下访问模式,求出该存储器读写时在存储器总线上的数据传输率。

(1) 全部访问为连续的读操作。

(2) 全部访问为连续的写操作。

(3) 65%的访问为读操作,35%的访问为写操作。

8. 假定在一个字长为32位的计算机系统中,存储器分别连接以下两种同步总线。

总线1是64位数据和地址复用的同步总线,能在一个时钟周期中传输一个64位的数据或地址。支持最多连续8个字的存储器读操作和存储器写操作总线事务,任何一个读写操作总是先用一个时钟周期传送地址,然后有两个时钟周期的延迟等待,从第4时钟周期开始,存储器准备好数据,总线以每个时钟周期两个字的速度传送,最多传送8个字。

总线2是分离的32位地址和32位数据的总线。支持最多连续8个字的存储器读操作和存储器写操作总线事务,读操作过程为:一个时钟周期传送地址,两个时钟周期延迟等待,从第4时钟周期开始,存储器准备好数据,总线以每个时钟周期一个字的速度传输最多8个字;对于写操作,在第一个时钟周期内第一个数据字与地址一起传输,经过两个时钟周期的等待延迟后,以每个时钟一个字的速度最多传输7个余下的数据字。

假定这两种总线的时钟频率都为100MHz,请回答以下问题。

(1) 两种总线的最大数据传输率(总线带宽)分别为多少?

(2) 连续进行单个字的存储器读操作总线事务时,两种总线的数据传输率分别是多少?

(3) 连续进行单个字的存储器写操作总线事务时,两种总线的数据传输率分别是多少?

(4) 每次传输8个字的数据块,其中,60%的访问是读操作总线事务,40%的访问是写操作总线事务,两种总线的数据传输率分别为多少?

（5）对上述各种计算结果进行分析后，你能得出什么结论？

9. 假定连接主存和CPU之间的同步总线具有以下特性：支持4字块和16字块（字长32位）两种长度的突发传送，总线时钟频率为200MHz，总线宽度为64位，每个64位数据的传送需一个时钟周期，向主存发送一个地址需要一个时钟周期，每个总线事务之间有两个空闲时钟周期。若访问主存时最初4个字的存取时间为200ns，随后每存取一个4字的时间是20ns，则在4字块和16字块两种传输方式下，该总线上传输256个字时的数据传输率分别是多少？你能从计算结果中得到什么结论？

10. 第9题所述的系统中，假定访问主存时最初4个字的读取时间为148ns，随后每读一个4字的时间为26ns，则在4字块和16字块两种传输方式下，CPU从主存读出256个字时，该总线上的数据传输率分别是多少？与上题计算结果进行比较分析，并给出相应的结论。

第 9 章 输入输出组织

输入输出组织主要用于控制外设与内存、外设与 CPU 之间进行数据交换。它是计算机系统中重要的软、硬件结合的子系统。通常把外部设备及其接口线路、I/O 控制部件以及 I/O 软件统称为输入输出系统。输入输出组织要解决的问题是对各种形式的信息进行输入和输出的控制。实现输入输出功能的关键是要解决以下一系列的问题：如何在 CPU、主存和外设之间建立一个高效的信息传输“通路”；怎样将用户的 I/O 请求转换成对设备的控制命令；如何对外设进行编址；怎样使 CPU 方便地寻找到要访问的外设；I/O 硬件和 I/O 软件如何协调完成主机和外设之间的数据传送等。

本章将围绕以上这些问题，重点介绍常用的外部设备、I/O 接口的功能和结构、外部设备的编址和寻址以及在主机和外设间进行数据传送的各种输入输出控制方式等内容。

9.1 外部设备的分类与特点

输入输出设备（又称外围设备或外部设备，简称外设）是计算机系统与人或它机之间进行信息交换的装置。输入设备的功能是把数据、命令、字符、图形、图像、声音或电流、电压等信息，以计算机可以接收和识别的二进制代码形式输入到计算机中，供计算机进行处理。输出设备的功能是把计算机处理的结果，变成人最终可以识别的数字、文字、图形、图像或声音等信息，然后播放、打印或显示输出。

9.1.1 外设的分类

外设按信息的传输方向来分，可分成输入设备、输出设备与输入输出设备三类。

（1）输入设备：包括键盘、鼠标、触摸屏、跟踪球、控制杆、数字化仪、扫描仪、手写笔、纸带输入机、卡片输入机、光学字符阅读机等。这类设备又可分成两类：媒体输入设备和交互式输入设备。媒体输入设备有光学字符阅读机、扫描仪等，这些设备把记录在各种媒体上的信息送入计算机，一般采用成批输入方式，一次成批输入一块数据，输入过程中不需操作者干预，因此这类设备属于成块传送设备；交互式设备有键盘、鼠标、触摸屏、手写笔、跟踪球等。这些设备由操作者通过操作直接输入信息。

（2）输出设备：包括显示器、打印机、绘图仪等。将计算机输出的数字信息转换成模拟信息，送往自动控制系统进行过程控制的数模转换设备也可以视为一类输出设备。

（3）输入输出设备：包括磁盘驱动器、磁带机、光盘驱动器、CRT 终端、网卡之类的通信

设备等。这类设备既可以输入信息,又可以输出信息。

外设按功能来分,可分成人机交互设备、存储设备和机-机通信设备三种。

(1) 人机交互设备:用于用户和计算机之间交互通信的设备。如键盘、鼠标、显示器、打印机等。大多数这类设备与主机交换信息以字符为单位,所以又称为字符型设备,或面向字符的设备。

(2) 存储设备:这类设备用于存储大容量数据,作为计算机的外存储器使用。如磁盘驱动器、光盘驱动器、磁带机等。这类设备与主机交换信息时采用成批方式,以几十、几百甚至更多字节组成的信息块为单位,因此属于成块传送设备。

(3) 机-机通信设备:主要用于计算机和计算机之间的通信,如网卡、调制解调器、数/模和模/数转换设备等。

当然,外设的分类还有其他方式,例如,按所处理信息的形态来分,可分成处理数字和文字的设备、处理图形与图像的设备以及处理声音与视频的设备等,这里不再赘述。

9.1.2 外设的特点

外设种类繁多,性能各异,但归纳起来有以下几个特点。

(1) 异步性:外设与 CPU 之间是完全异步的工作方式,两者之间无统一的时钟,且各类外设之间工作速度相差很大,它们的操作在很大程度上独立于 CPU,但又要在某个时刻接受 CPU 的控制,这就势必造成输入输出操作相对 CPU 时间的任意性与异步性。必须保证在连续两次 CPU 和外设交往之间,CPU 仍能高速地运行它自己的程序,以达到 CPU 与外设之间、外设与外设之间的并行工作。

(2) 实时性:一个计算机系统中,可能连接了各种各样类型的外设,且这些外设中有慢速设备,也有快速设备,CPU 必须及时按不同的传输速率和不同的传输方式接收来自多个外设的信息或向多个外设发送信息,否则高速设备可能有丢失信息的危险。

(3) 多样性:由于外设的多样性,它们的物理特性差异很大,信息类型与结构格式多种多样,这就造成了主机与外设之间连接的复杂性。为简化控制,计算机系统中往往提供一些标准接口,以便各类外设通过自己的设备控制器与标准接口相连,而主机无须了解各特定外设的具体要求,可以通过统一的命令控制程序来实现对外设的控制。

9.2 输入设备和输出设备

最常用的输入设备是键盘与鼠标,相对而言它们比较简单。最常用的输出设备是打印机与显示器。下面对它们作简略介绍。

*9.2.1 键盘

键盘是计算机不可缺少的最常用的输入设备,用户通过键盘可向计算机输入字母、数字和符号。键盘有外壳、按键和电路板三部分组成。按键的结构可归纳成两类:一类是触点式按键,它借助触点开关的接通或断开,来产生电信号;另一类是非触点式开关,利用电压、电流或电磁场的变化产生输出信号。计算机中基本上使用后一类开关。电路板中是键盘控制逻辑,由一些逻辑电路或单片机组成,键盘上每个按键发出的信号由电路板转换为二进制

代码，然后通过键盘接口送入计算机中。

按键的排列是一个 $m \times n$ 的二维阵列，每个按键对应该阵列中的一个位置。键盘采用“按列扫描、接地检查”的方式进行工作。具体过程如下：键盘电路板内的单片机每隔 3～5ms 对按键矩阵的各列进行顺序扫描，被扫描列接 0 电平“地”，其他列接高电平。若此时被扫描的列中正好有某个键被按下，则相应的行和列被接通，因而按键所在行输出变低电平，其他行输出为高电平。如图 9.1 所示，当扫描最右边一列时，该列第二行的“A”键被按下，所以，第二行的输出为低电平 0。此时，单片机可根据扫描的列号和输出为 0 电平的行号得到按键的位置，位置信息称为按键的“扫描码”或“位置码”。

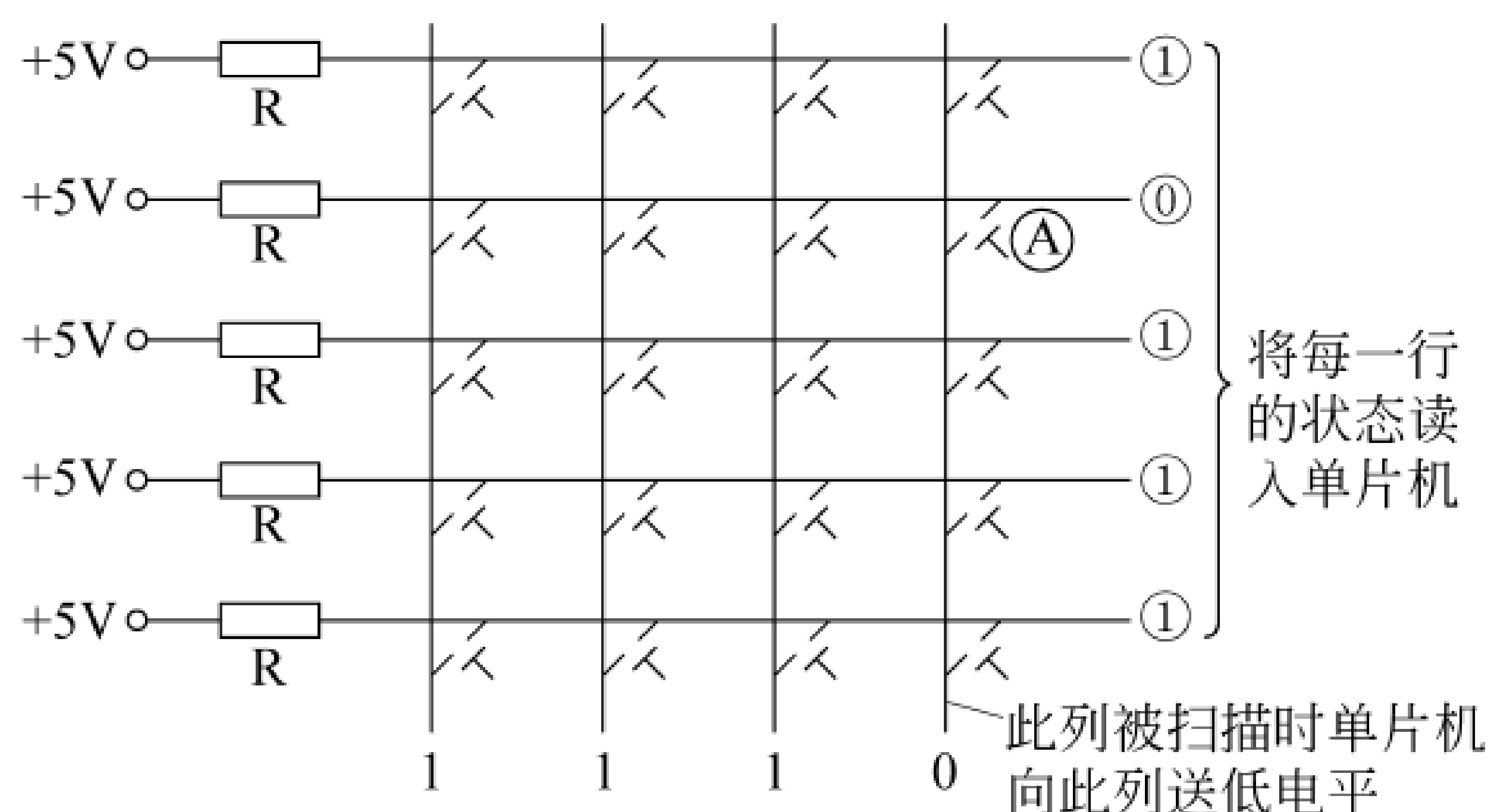


图 9.1　按键矩阵与扫描码的形成

根据键盘输入主机的信息的不同，键盘分成编码键盘和非编码键盘两类。

（1）编码键盘：键盘电路板中有一个编码器，能将单片机得到的位置码转换成相应的 ASCII 码送入主机。编码器由硬件构成，因此，键盘响应速度快，但键盘控制电路的结构要复杂一些。

（2）非编码键盘：键盘送到主机的信息是位置码，由键盘中断服务程序完成将位置码转换成 ASCII 码。这种键盘控制电路较简单，当然速度会慢一些。但是，由于主机速度远比人按键速度快，且这种键盘上某些键的功能可以通过软件来重定义，使用灵活，因此，目前 PC 大多用这种键盘。

键盘中的单片机除了完成按键扫描和生成扫描码的功能之外，还将扫描码转换成串行形式发送给主机，并具有消除抖动、扫描码缓冲和自动重复等功能。

键盘发出的串行数据由 1 位起始位、8 位数据位、1 位奇偶校验位和 1 位停止位组成。主机的键盘控制电路接收到这些串行数据后，去掉起始位、校验位和停止位，将 8 位数据通过串-并转换，形成并行数据送入缓冲寄存器，然后向 CPU 发出键盘中断请求，CPU 响应该中断后，由键盘中断服务程序把扫描码转换成 ASCII 码，然后送入主存中的键盘数据缓冲区。

键盘上也可输入如汉字等非西文字符，这由各种汉字输入法自行定义。键盘位置码送入计算机后，经过汉字输入软件的处理，转换成该汉字对应的内码，再进行显示、存储等其他操作。

键盘和主机的接口有 AT 接口、PS/2 接口和 USB 接口三种。较早的机器采用 AT 接口（大五芯接口），现在台式机大多用 PS/2 接口（小五芯接口）或 USB 接口。USB 接口支持即插即用，因而使用方便，比较受欢迎。

*9.2.2 鼠标器

鼠标器(mouse)是一种相对定位设备。它能方便地控制屏幕上的光标移动到指定的位置,并通过按键完成各种操作。鼠标器由于其外形如老鼠而得名,通过电缆与主机相连接。鼠标器在桌上移动,其底部的传感器检测出运动方向和相对距离,送入计算机。

根据鼠标器所采用传感器技术的不同,鼠标器可以分成两类:机械式与光电式。

(1) 机械式鼠标器:其底部有一个圆球,鼠标移动时,圆球滚动带动与球相连的圆盘。圆盘上的编码器把运动方向与距离送给主机,经软件处理,控制光标作相应移动。该类鼠标器简单,使用方便,但也容易磨损,且精度差。

(2) 光电式鼠标器:几乎没有任何机械零件,而是使用一个微型光学镜头不断地拍摄鼠标下方的图像,数字信号处理器(DSP)对获取的图像序列中帧与帧之间的变化进行分析,计算出移动方向和距离,送入计算机,控制光标的移动。这类鼠标器速度快,精度高,没有机械磨损,使用寿命长,不需鼠标垫,只要在平面上就能操作,是目前比较流行的鼠标器类型。

鼠标器的技术指标之一是分辨率,分辨率越高,越有利于用户的细微操作。分辨率用dpi(dots per inch)表示,它指鼠标在桌上每移动一英寸,光标在屏幕上所移动的像素数。对光电鼠标来说,反映其性能的另一个指标是帧速率,即刷新频率。它指 DSP 每秒钟可以处理的图像帧数。

鼠标器与主机相连的接口主要有 USB 或 PS/2 接口。早期的个人计算机把鼠标器接在串行通信口 COM1 或 COM2 上,目前已很少这样使用了。

*9.2.3 打印机

打印机是计算机系统中最基本的输出设备。目前使用的打印机主要有针式打印机、激光打印机和喷墨打印机三种。

1. 针式打印机

击打式打印机是最早研制成功的计算机打印设备。它以机械力量击打字锤从而使字模隔着色带在纸上打印出字来。按字锤或字模的构成方式来分,又可以分成整字形打印设备和点阵打印设备两类。整字形击打设备利用完整字形的字模每击打一次印出一完整字形。这类设备的优点是印字美观自然,可同时复印数份。缺点是噪音大,印字速率低,字符种类少,无法打印汉字或图形,且易磨损,目前已很少使用;点阵式击打设备是利用打印头中的多根印针经色带在纸上打印出点阵字符的印字设备,又称针式打印机。目前有 7 针、9 针、24 针或 48 针的印字头。这类打印设备相对于整字型击打设备,其机械结构简单,印字速度快,噪声小,成本低,目前在银行、证券等行业的票据打印中广泛使用。以下主要介绍点阵打印机的原理。

点阵打印机有串行打印机和并行打印机两种。这里以串行打印机为例来说明其打印原理。

假定是 24 针打印机,即一列印针为 24 根,其印字头上有一列 0.2～0.3mm 直径的印针及其驱动电磁铁,印字头自左至右按点距移动。驱动印针的电磁铁受字符点阵内容的控制:有点的地方即打印,无点的地方不打印。每完成一列打印,印字头右移一个点距。如此一列一列地打印,直到一行打印结束,打印机走纸一行的距离,印字头再返回起始位置重新开始

打印。其输出数据与一列印针之间的对应关系如图 9.2 所示。

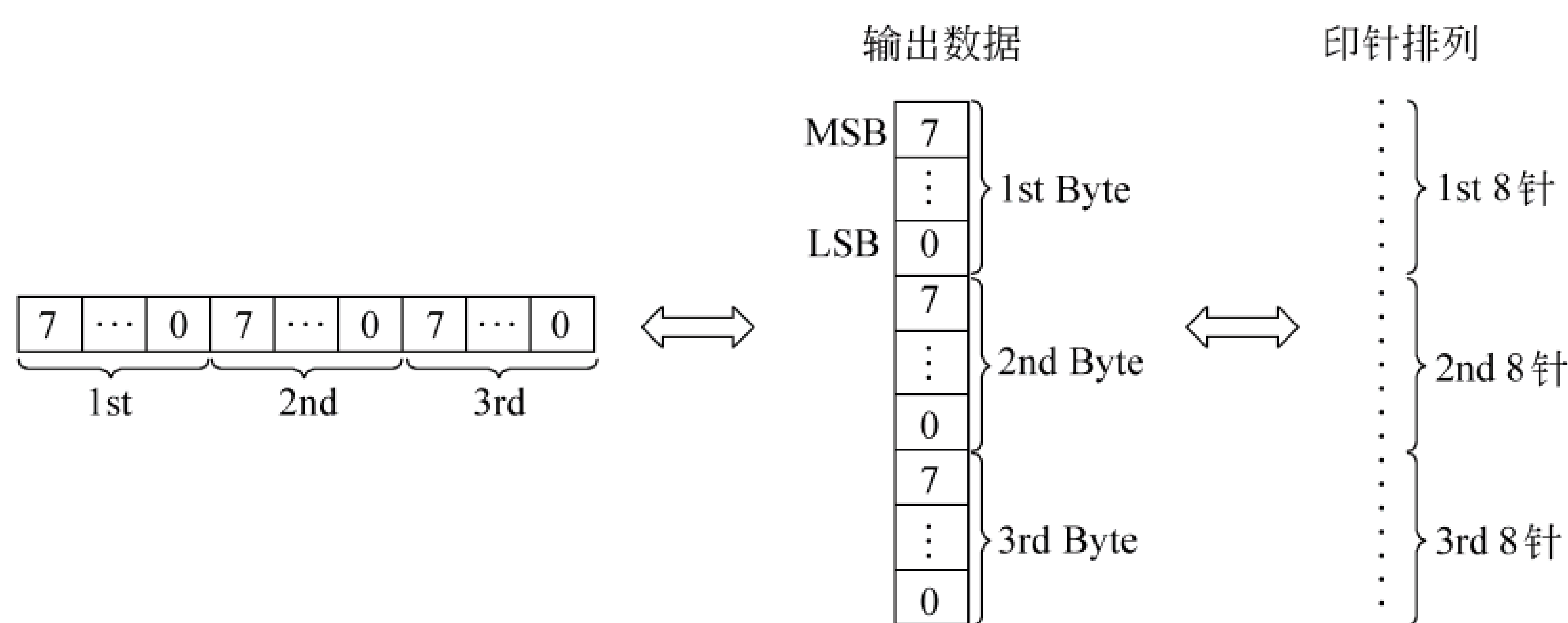


图 9.2　数据与印针的对应关系

打印驱动程序从主存读出输出点阵数据到打印缓存中。每读出三个字节，对应 24 针的一列打印数据。从图 9.2 中可看出，打印数据在打印缓存中是按列存放的，因此，在送数据到打印缓存之前，需要将打印字符的字模点阵数据进行行和列的转换。如此每三个字节地读出数据送打印缓冲，直到一行读出完毕，然后发回车或换行命令，打印机才真正把打印缓冲中的一行数据在纸上打印出来。这里必须注意的是输出字符点阵或图形点阵与打印针之间的对应关系，包括高/低字节的安排，不同类型的打印机，其排列是有差别的。

打印机通过打印控制器或打印适配器与主机连接，打印控制器由以下基本部件组成。

(1) 数据锁存器：暂存 CPU 送来的打印数据。该数据可以送打印机缓冲器准备打印，也可以回送 CPU 以便进行检测用。

(2) 命令译码器：对 CPU 送来的命令进行译码，产生打印控制器内部使用的几个命令，如数据传输方向、读数据、写数据、读控制、写控制和读状态等。

(3) 控制锁存器：锁存 CPU 送来的控制命令，如初始化、选通、自动走纸等命令。这些命令也可以回送 CPU 以便检测用。

(4) 状态锁存器：保存打印机送来的状态信息，如打印机忙、缺纸、联机、认可和出错等，以供 CPU 随时检测用。

图 9.3 是打印机与主机的连接示意图。打印控制器与 CPU 之间的连接有数据线(发送或接收数据、状态和命令字节)、地址线(选择打印控制器中寄存器的地址)和中断请求信号 INT 等。打印控制器与打印机之间若用 25 芯并口连接的话，则有 8 位数据线 D0～D7，打印控制器送给打印机的初始化、选通、自动走纸等命令线以及打印机回送打印控制器的反映打印机目前状态的忙、缺纸、联机、出错、认可等信号线。

打印机初始化主要是清除打印机缓冲和初始化打印机动作(如打击头回到起始位置等)。CPU 输出一个字符后，要判断打印机状态“忙”否。只有当打印机不忙时，CPU 发送选通信号，才把打印数据真正送到打印机缓冲。当打印机缓冲中接收满一行数据后，CPU 发回车或换行信号，打印机才真正完成打印动作。

打印机除了能接收主机送来的数字、字符等打印信息以外，它还能接收主机送来的控制打印机执行特定动作的命令，这些命令通常称为不可打印字符。这些特定字符分成两类。第一类是回车(CR)、换行(LF)、删除(DEL)、响铃(BELL)等使打印机执行某特定动作的字符，其值可见 ASCII 码表，不同打印机的码值不变；第二类是 ESC 命令序列，或称“换码”序

列。其命令格式中,第一个字符固定为ESC,后跟一个或多个ASCII字符。只有当打印机接收到一个完整的命令序列以后,才完成规定的控制功能。打印机不同,往往命令序列格式也不一样,因此,很多情况下,打印驱动程序是不兼容的。

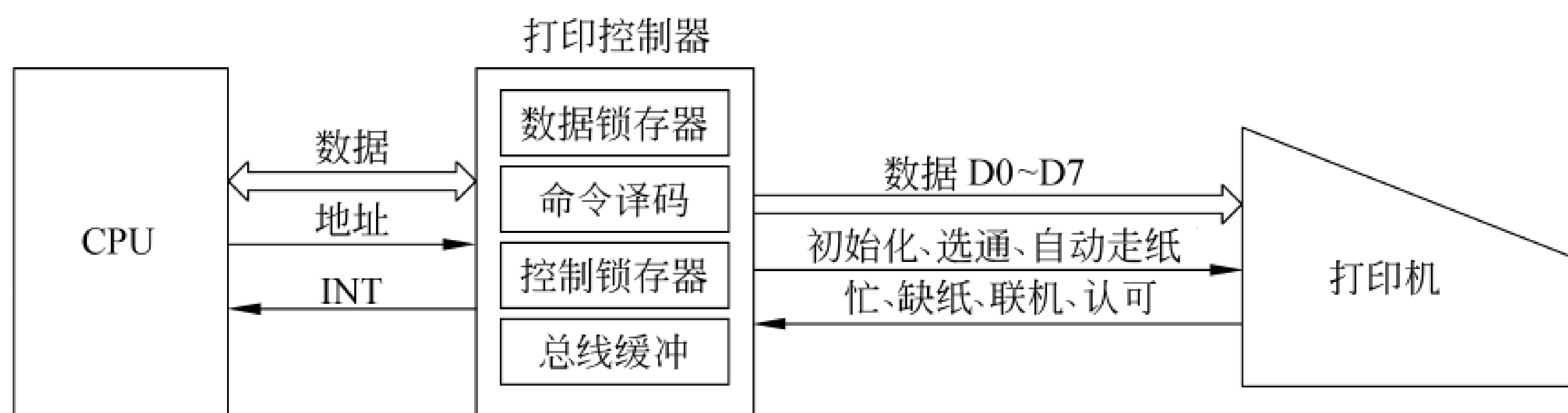


图9.3 打印机与主机的连接

2. 激光打印机

激光打印机印刷速度快、印字质量好、噪音低、分辨率高、印刷输出成本低,是目前应用最广泛的一种非击打式印字机。

激光印字机由打印机控制器和打印装置两部分组成。打印机控制器一般由功能较强的处理器、缓冲存储器以及相应的辅助电路构成,负责与主机的通信、解释主机送来的打印语言(包括打印机控制命令、页面格式命令、字体处理命令、图形命令等)、格式化打印内容(如纸张尺寸、边界设定、字符的大小与位置等)、光栅化处理(把经过格式化的页面数据转换成点阵数据)等,然后送打印装置进行输出。

如果是彩色激光印字机,一般带有多种颜色的硒鼓,最典型的是C(青色)、M(品红)、Y(黄色)、K(黑色)四种颜色。彩色印刷过程可先由处理器把彩色图像分解成C、M、Y、K四种单色的图像(称为"分色"过程),再由打印装置分四次套色印刷来完成。

打印语言是一组控制打印机工作的命令,打印机按照这些命令来处理主机送来的打印数据,并最终打印出复杂的文字与图像。打印语言大体分成两类:一类是页面描述语言(PDL),其中以Adobe公司的PostScript语言最为流行;另一类是Escape码语言,其中以HP公司的PCL语言最为流行。

(1) PostScript语言:诞生于20世纪80年代中期,它有很强的图形、图像、文字和彩色处理功能,目前广泛应用于高档彩色激光印字机、激光照排机等要求高质量、彩色印刷结果的场合。

(2) PCL语言:它是一种Escape码语言,前面讲到的点阵打印机中的Esc命令序列实际上是一种低版本的PCL语言。随着PCL语言版本的不断升级,其处理图形、图像、文字与彩色的功能日趋增强,且由于它比PostScript语言简单一些,解释速度快,所以目前广泛安装在激光印字机、喷墨打印机等设备中。

喷墨打印机也是一种非击打式打印机,它利用喷墨头喷射出可控的墨滴从而在打印纸上形成文字或图片,也是目前应用较多的一种打印输出设备。

过去,打印机与主机的接口主要是25芯的并行口,不支持即插即用,带电插拔时,一不小心就烧坏机器,很不方便。现在大多数激光打印机和喷墨打印机都已经采用USB接口和主机连接,高速打印机也可以用SCSI接口和主机相连。

*9.2.4 显示器

显示器是用来显示数字、字符、图形和图像的设备,它由显示器(也称监视器)和显示控制器组成,是计算机系统中最常用的输出设备之一。计算机使用的显示器主要分为两种:阴极射线管(CRT)显示器以及目前流行的液晶平板(LCD)显示器。

1. CRT 显示器

早期常用的显示器是 CRT 显示器。CRT 显示器由阴极射线管(CRT)、亮度控制电路(控制栅)以及扫描偏转电路(水平/垂直扫描偏转线圈)等部件构成,如图 9.4 所示。

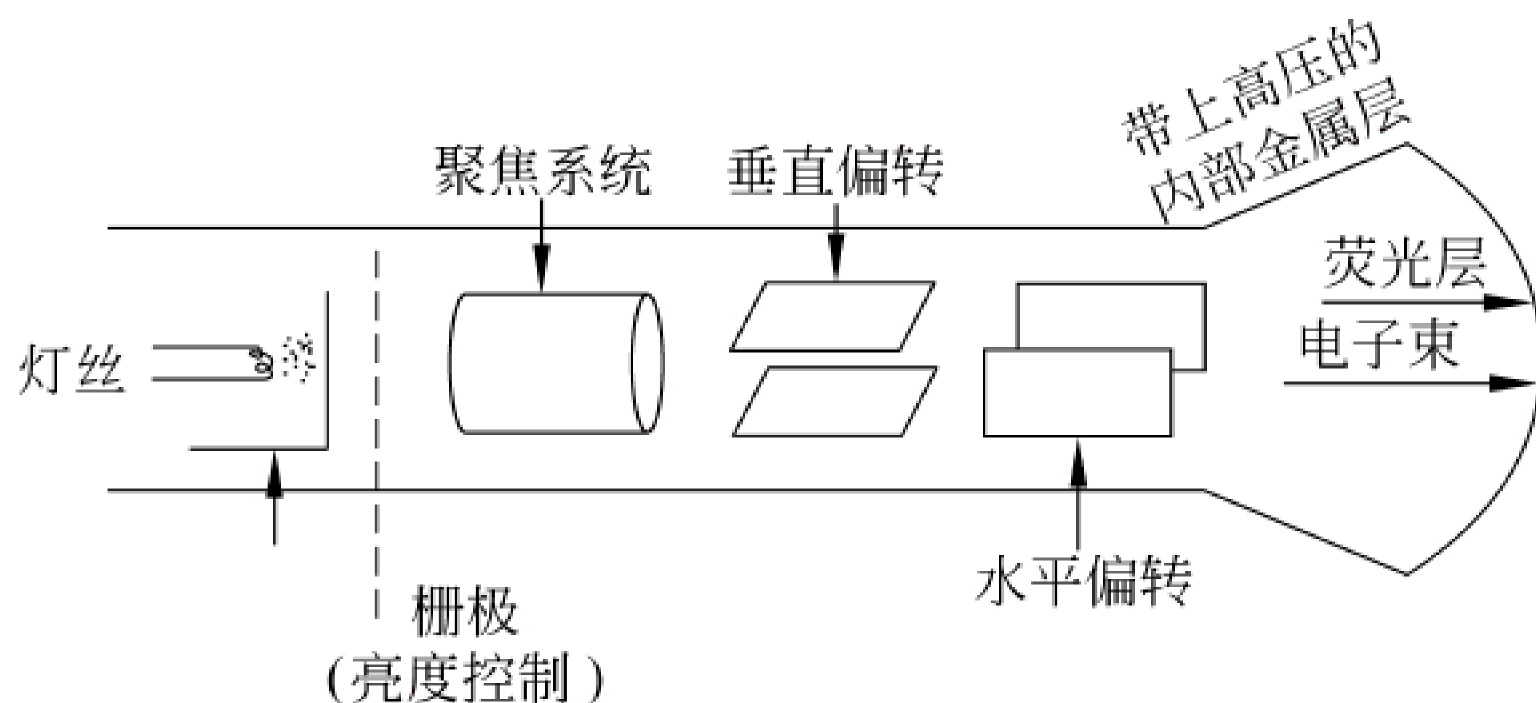


图 9.4 CRT 显示器的原理

由热发射产生的电子流在真空中在几千伏高压影响下射向 CRT 前部,控制栅的电压决定有多少电子被允许通过,经过聚焦的电子束在水平与垂直偏转电路控制下射向屏幕,轰击涂有荧光粉的 CRT 屏幕,产生光点。通过对控制栅电压强弱的控制,达到控制光点有无的目的,从而形成显示图像。

CRT 显示器采用光栅扫描的方式显示图像。电子束在水平同步信号和垂直同步信号的控制下,在屏幕上按行优先的方式形成光点。从左向右进行行扫描,每扫描完一行,就进行一次水平行回扫,回到最左边,再进行下一行水平扫描,所有行扫描后形成一帧图像,然后进行垂直回扫,回到左上角,重复进行下一帧图像的扫描。水平扫描周期的倒数称为行频,垂直扫描周期的倒数称为帧频,也称刷新频率。

CRT 显示器的一个缺点是会出现屏幕闪烁。为了保证屏幕上显示的图像不产生闪烁,刷新频率必须达到 50～70Hz,最好在 75Hz 以上。固定分辨率的图形显示器的行频、水平扫描周期、每像素的读出时间等,均有一定要求。例如,当分辨率为 640×480,刷新频率为 50Hz 时,且假定水平回扫期和垂直回扫期各占水平扫描周期和垂直扫描周期的 20%,则得到以下相关参数。

行频:480 线×50 帧/秒÷80%=30kHz。

水平扫描周期:1/30kHz=33μs。

每一像素的读出时间:33μs×80%÷640=41ns。

若分辨率提高到 1024×768,帧频为 60 帧/秒,则行频应该提高到 57.6kHz,水平扫描周期 HC=17.4μs,每像素读出时间减少到 13.6ns。显然,分辨率越高,为保证图像不闪烁,时间要求越高(每一像素的读出和显示时间越短),成本也随之上升。光栅扫描显示器的扫描方式可分成逐行扫描与隔行扫描方式两种。

CRT 显示器有三个电子枪,射出的电子流必须精确聚集,否则就得不到清晰的图像显

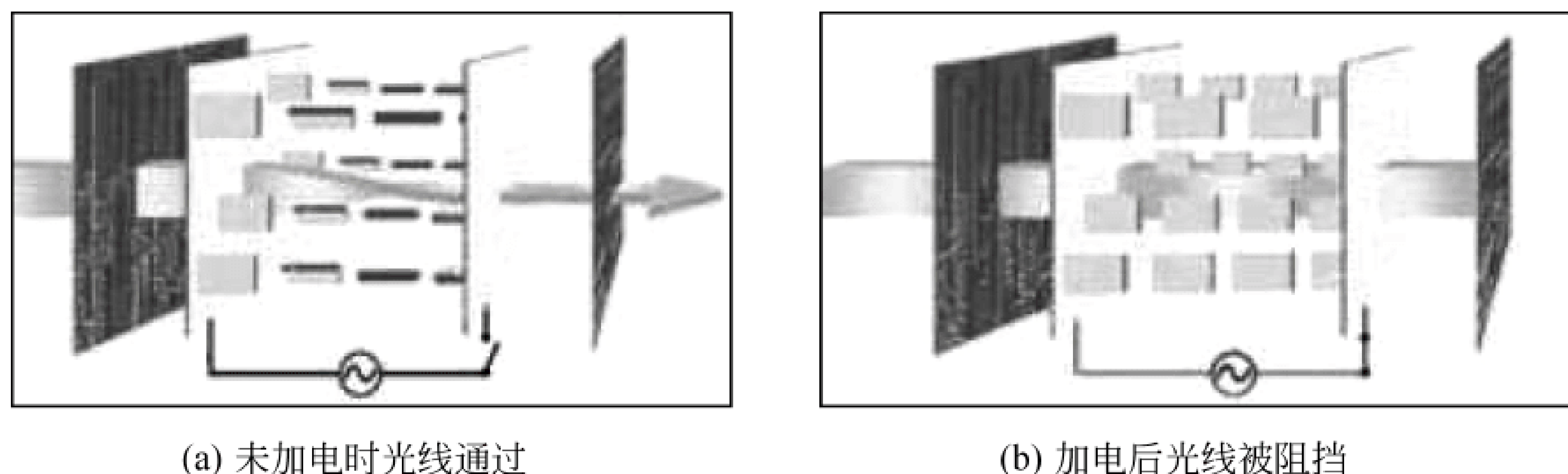
示。因此，CRT 显示器可能会因聚焦不准而造成图像模糊；此外，它还有体积大、能耗高的缺点。目前 CRT 显示器已经逐步为液晶显示器所取代。

2. 液晶显示器(LCD)

液晶显示器的基本原理是基于液晶如下的物理特性：液晶通电时会改变其排列次序，从而影响光线的通过。

如图 9.5 所示，液晶面板包含了两片偏振方向相互垂直的偏振光过滤片，中间夹着一层液晶，液晶层的两侧分别连接着一个带有精细开槽(透明电极)的玻璃基板，这两个基板上的槽互相垂直。液晶由长棒状的分子构成，将液晶倒入精细开槽平面后，液晶分子会顺槽排列。因此，两个基板之间的液晶分子在第一个上呈垂直排列而在第二个上呈水平排列；在两侧相互垂直排列的限制下，中间部分的液晶分子在自身电荷的作用下被逐步扭转排列成一个 90°的螺旋。当光束从第一个垂直光偏振板进入后通过液晶层时，垂直偏振光将顺着液晶的螺旋被偏转 90°成为水平偏振光，因此顺利通过第二个水平偏振光过滤片(图 9.5(a))。

如果在液晶的两个透明电极上通电，在电场电荷的作用下，原来螺旋排列的液晶分子将重新排列成一致的方向，不再扭转原垂直偏振光的方向；垂直偏振光线到达第二个水平偏振光过滤片时，将受到阻挡而不能通过(图 9.5(b))。调节电压的大小可以改变液晶偏转的角度，进而可以改变通过光线的旋转幅度，最终控制光线通过的亮度。



(a) 未加电时光线通过　　(b) 加电后光线被阻挡

图 9.5　液晶显示基本原理图

也可以通过改变 LCD 中的液晶排列方式，使光线在加电时射出，而不加电时被阻挡。但由于计算机的屏幕几乎总是亮着的，所以只有“加电将光线阻挡”的方案才能达到最省电的目的。

为了实现彩色显示，还要加上专门处理彩色显示的彩色滤光片。通常，在彩色 LCD 面板中，每一个像素由 3 个液晶单元格构成，每一个单元格前分别有红(R)、绿(G)、蓝(B)彩色滤光片。这样，通过不同单元格的光线就可在屏幕上显示出不同的颜色。

因为每个液晶单元都是独立开和关，不存在聚焦问题，因此，LCD 屏幕上图像非常清晰；同时，LCD 显示器不需要采用 CRT 显示器那样的光栅扫描，因此也没有屏幕闪烁问题。由于具有体积小、耗电低、不闪烁等优点和良好的综合性能，LCD 显示器目前已广泛应用于便携式计算机、数码相机和电视机等设备。当然，LCD 显示器也存在价格高、视角不广以及彩色显示不够鲜艳等缺点。

通常显示器工作时有两种模式。一种是字符模式，显示存储器(简称显存、VRAM，也称刷新存储器)中存放的是字符的编码(ASCII 码或汉字代码)及其属性(如加亮、闪烁等)，其字形信息存放在字符发生器中。另一种模式是图形模式，此时每一字符的点阵信息直接

存储在显示存储器中，字符在屏幕上的显示位置可以定位到任意点。

彩色或单色多级灰度图像显示时，每一个像素需要使用多个二进位来表示，每个像素对应的二进位数称为颜色深度。例如，若颜色深度为 8bit，则可以有 256 色；若颜色深度为 24bit，则可达 16M 种颜色(称为真彩色)。在图形显示模式下，显示控制器除了完成前述的两个基本功能外，还能实现画图功能。显示控制器接收并实现 CPU 送来的画图命令，并将结果写入显存；同时，显示控制器读出显存内容，经并/串变换和 D/A 转换后，将 R、G、B 三个不同的颜色控制信号送显示器，从而在屏幕上显示出彩色图形，显示控制器的功能越来越强，除了完成二维画图命令以外，还具有三维图形显示功能。它可以集成在主板上，也可以以图形显示卡(简称显卡)的方式插在主板扩充槽中。

显卡的核心是绘图处理器(Graphics Processing Unit，GPU)。早期的绘图功能都由 CPU 在内存中完成，然后将生成的图像位图从内存传送到显存中。这种方案显然加重了 CPU 的工作量，并且绘图速度慢。目前显卡中的 GPU 专门用来进行绘图，它有一组可高速执行的适用于图像和图形处理的指令，如数据块传送、基本图形绘制、区域填空、图案填空、图形缩放、颜色转换等。由于采用专用处理器实现，所以图形操作速度快，并且大大减轻了 CPU 负担。

9.3 外部存储设备

9.3.1 磁表面存储原理

磁表面存储器包括磁鼓、磁带、磁盘和磁卡片等。目前，在计算机系统中以磁盘和磁带为主。磁表面存储器主要用作计算机存储系统中的辅存，它可以存储大量的程序和数据。

1. 磁层和磁头

磁表面存储器中信息的存取，主要由磁层和磁头来完成。磁层是存放信息的介质，它由非矩形剩磁特性的导磁材料(如氧化铁、镍钴合金等)构成。将这种材料制成的磁胶涂敷或镀在载磁体上，其厚度通常为 0.1～5μm，以记录信息。载磁体可以是金属合金(硬质载磁体)或者是塑料(软质载磁体)。

为获得良好的技术性能，磁层材料的剩磁(BR)要大，矫顽力(HC)要合适，才能有足够的抗干扰能力和使用较小的写电流。磁层厚度要薄，才能提高记录密度。此外对生成磁层的工艺、机械性能等也有一定的要求。

磁头是实现“磁-电”和“电-磁”转换的元件。它是由高导磁率的软磁性材料做成铁芯，在铁芯上开有缝隙并绕有线圈。当载磁体与磁头作相对运动时，若写磁头线圈通以磁化电流，则可将信息写到磁层上。当读磁头通过磁层上某一磁化单位而形成磁通回路时，磁通的变化使线圈两端产生感应电势，形成读出信号。

磁头质量的好坏，不仅与铁芯的材料有关，而且与磁头加工工艺有关。如磁头中的缝隙形状和尺寸将直接影响记录密度和读出幅度。磁头一般分读磁头、写磁头和读写磁头，其铁芯上的线圈分别称为读线圈、写线圈和读写线圈。

2. 磁表面存储器的读写过程

在磁表面存储器中，一般都使磁头固定，而磁层(载磁体)作高速回转或匀速直线运动。

在这种相对运动中，通过磁头（其缝隙对准磁层）进行信息存取。

（1）信息写入过程

磁头写线圈中通以写电流脉冲，此电流在铁芯中生成磁场，其磁头缝隙处的磁场穿过磁层中一微小区域，使该区域磁层以一定方向被磁化，形成一个磁化单元，而写电流脉冲消失后，磁层仍保持该方向的剩磁状态＋BR或－BR，利用这两种稳定的剩磁状态，可以表示二进制代码的0和1。显然。一个磁化单元就是一个存储元，存储一位二进制信息。当磁层相对于磁头运动时，就可以连续写入一连串的二进制信息。这就是信息的写入过程。

（2）信息读出过程

读出时，磁头与磁层同样作相对运动。当磁层中某一记录单元运动到磁头缝隙下面时，由于磁头铁芯是良好的导磁材料，磁化单元的磁力线很容易通过磁头而形成闭合磁通回路。由于磁头中的磁阻比其周围空气的磁阻小得多，便在磁头中产生较大的磁通变化，因而在读线圈的两端产生较大的感应电势E，经读出放大电路整形和放大后成为读出信号。由于不同极性的磁化单元在铁芯中的方向不同，因而形成的感应电势E的方向也不同，从而可区别读出的是0还是1，图9.6是一段磁化单元的读出过程示意图。

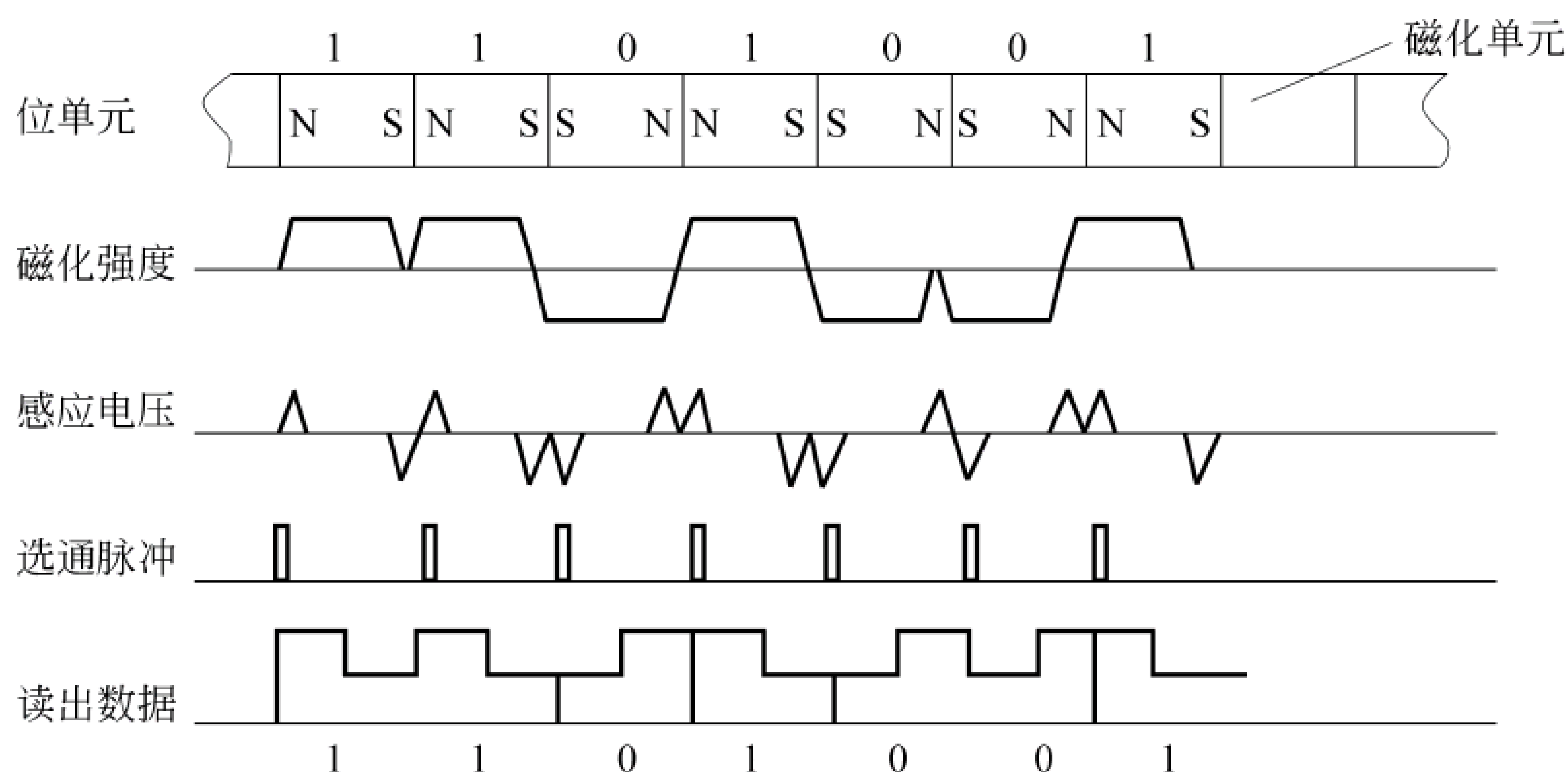


图9.6　磁化单元的读出过程

3．磁表面存储器的性能指标

（1）记录密度

记录密度可用道密度和位密度来表示。磁道是在磁层运动方向上被磁头扫过的轨迹。一个磁表面会有许多磁道。在沿磁道分布方向上，单位长度内的磁道数目，叫道密度。常用的道密度单位为tpi（每英寸磁道数）和tpm（每毫米磁道数），目前软磁盘的道密度约为40～150tpi。硬盘的道密度高得多。在沿磁道方向上，单位长度内存放的二进制信息的数目叫位密度。常用的位密度单位为bpi（每英寸二进制位数）和bpm（每毫米二进制位数）。

如图9.7所示是磁盘盘面上的道密度和位密度示意图。左边采用的是低密度存储方式，所有磁道上的扇区数相同，所以每个磁道上的位数相同，因而内道上的位密度比外道位密度高；右边采用的是高密度存储方式，每个磁道上的位密度相同，所以外道上的扇区数比内道上扇区数多，因而整个磁盘的容量比低密度盘高得多。

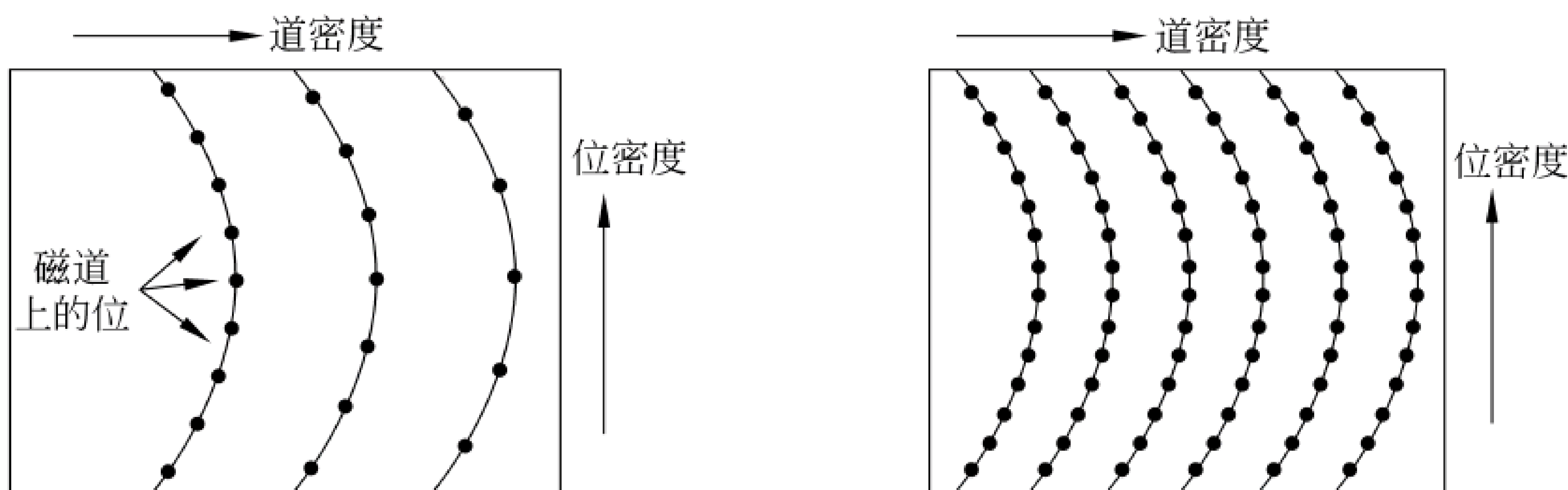


图 9.7　磁盘盘面上的记录密度示意图

(2) 存储容量

存储容量指整个存储器所能存放的二进制信息量。它与磁表面大小和记录密度密切相关。

(3) 平均存取时间(average access time)

磁表面存储器的读写是在磁层相对磁头作匀速运动的过程中完成的。由于主机对存储器读写数据的随机性,很难保证磁层上所需要读写的数据块位置正好处于磁头下方。因此,存取时间应包括磁头定位和数据传输等多个部分的时间。

以磁盘为例,磁头定位过程包括寻道和旋转等待两个阶段,因此,磁头定位所用的时间包括寻道时间(seek time)和旋转等待时间(rotational latency)。寻道时间是指磁头从读写前原有的位置移到指定读写的磁道上所花费的时间。旋转等待时间是指磁头进入指定磁道后,该磁道上被存取的信息段正好旋转到磁头下方所需的时间。

磁带的定位时间与走带速度和带长有关,磁带愈长,带速愈慢,其平均时间愈长。

(4) 数据传输速率

数据传输速率是指磁表面存储器完成磁头定位和旋转等待以后,单位时间内从存储介质上读出或写入的二进制信息量。为区别于外部数据传输率,通常称之为内部传输速率(internal transfer rate),也称为持续传输速率(sustained transfer rate)。而外部传输速率(external transfer rate)是指主机中的外设控制接口从(向)外存储器的缓存读出(写入)数据的速度,由外设采用的接口类型决定。通常称外部传输速率为突发数据传输速率(burst data transfer rate)或接口传输速率。

4. 数据记录方式

数据记录方式是指将数字信息转换成磁层表面的磁化单元所采用的各种方式。由于磁化过程是通过在磁头中通以磁化电流来实现的,故记录方式取决于写电流波形的组合方式。记录方式的选取将直接影响到记录密度、存储容量、传送速率以及读写的控制逻辑。

数据记录方式按照写信息所施加的电流波形的极性、频率和相位的不同,分为归零制、不归零制、调相制和调频制等。

(1) 归零制(RZ 制)

写“1”用正脉冲,写“0”用负脉冲,一位信息写完后,电流总回归到零。这种记录方式又叫双向归零制或典型归零制。磁化单元的剩磁方向在存“1”时为+Br,存“0”时为-Br。归零制的写电流波形如图 9.8 所示,其主要特点如下。

写入前先退磁，两个信息位之间有未磁化的间隙（B＝0），记录密度较低。每个位单元有两个读出波形，具有自同步能力，即能从本磁道读出的信息脉冲序列中提取出选通时钟信号，而无须增加附加的同步磁道。

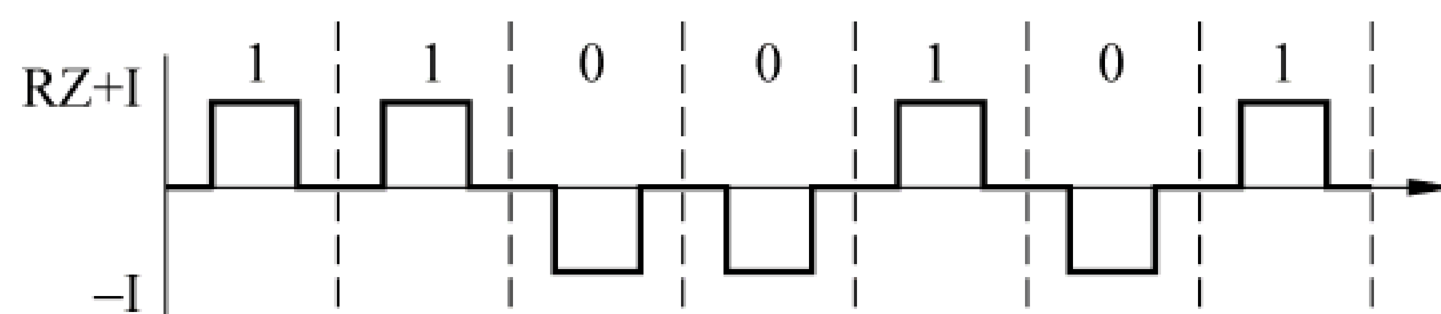


图 9.8　归零制写电流波形

（2）不归零－1 制（NRZ－1 制）

写电流只在写“1”时改变方向，写“0”时写电流不变，所以又称作“见 1 就翻”不归零制。各信息位间无“间隙”，记录密度较高。存“1”才能读出信号，存“0”无读出信号，故无自同步能力。由于电流不回到零，功耗较大，写电流波形如图 9.9 所示。

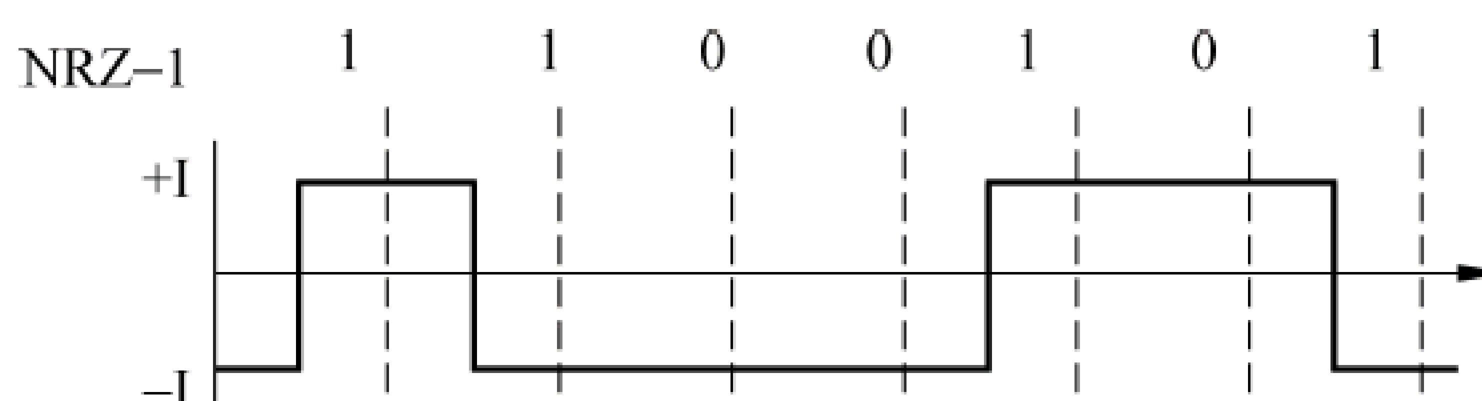


图 9.9　归零－1 制写电流波形

（3）调相制（PM 制）

它是利用写电流的相位不同实现写“1”和写“0”的一种记录方式。写“0”时，写电流先正后负；写“1”时，写电流是先负后正，调相制波形如图 9.10 所示。

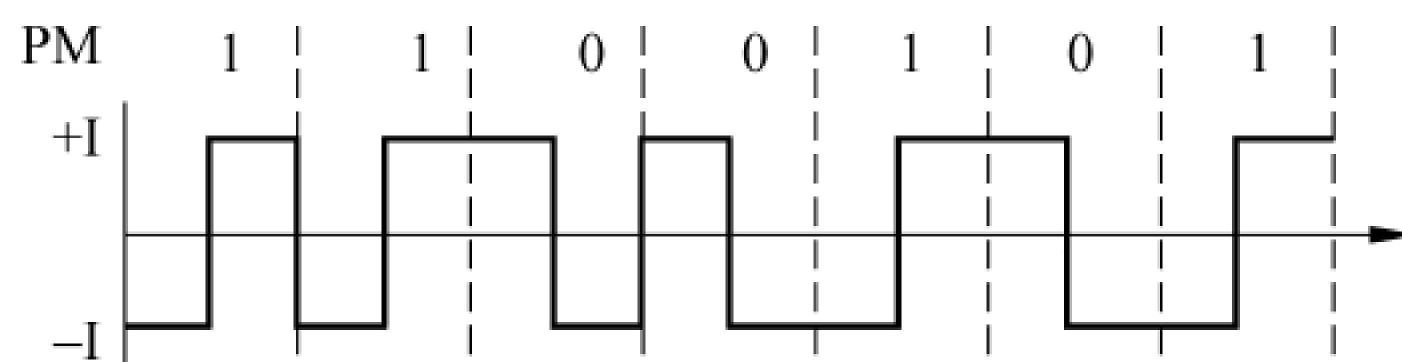


图 9.10　调相制写电流波形

此方式的主要特点是，无论写“1”还是写“0”，在一个位信息期间，写电流相位至少有一次改变。因此，它和下面要叙述的调频制有类同性，利用记录信号变向，可生成读同步脉冲。

（4）调频制（FM 制）

写入“0”和“1”时磁头所加的写电流的频率不同，其写电流波形如图 9.11 所示。调频制的主要特点有三个：①每记录一个代码时，在两个信息位的交界处，写电流一定改变方向。②写“1”时，写电流的频率比写“0”时的频率高一倍。③有自同步能力。

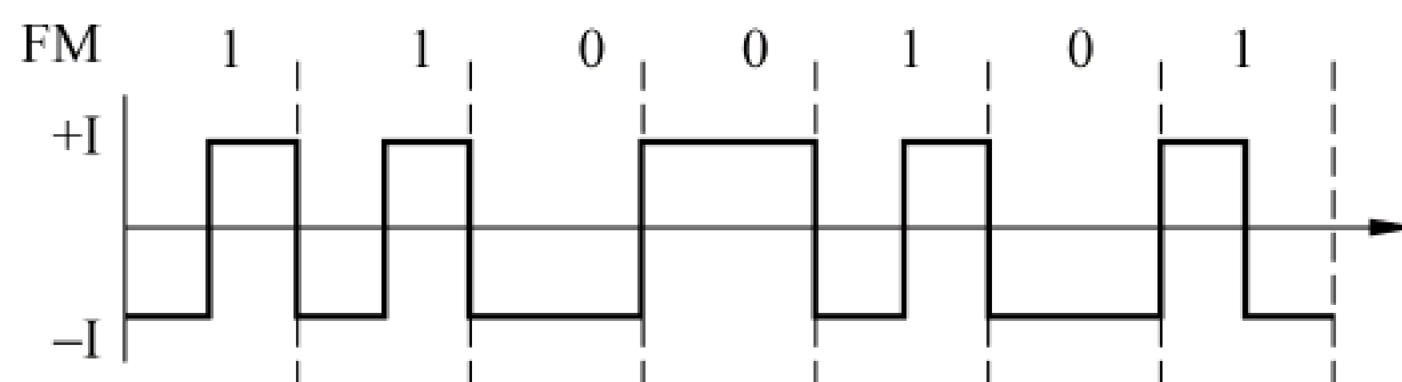


图 9.11　调频制写电流波形

(5) 改进调频制(MFM 制)

改进调频制和调频制的区别在于去掉了 FM 制中的冗余信息,其特点是:a)写电流不是在每个位周期的起始处都翻转,而只有连续记录两个或两个以上"0"时,才在位周期的起始处翻转一次;b)逢 1 在位中央翻转一次。这样,仍保持了自同步能力,记录密度又得到了提高,故又称倍密度记录方式,在磁盘中得到广泛应用,图 9.12 是其写电流波形。

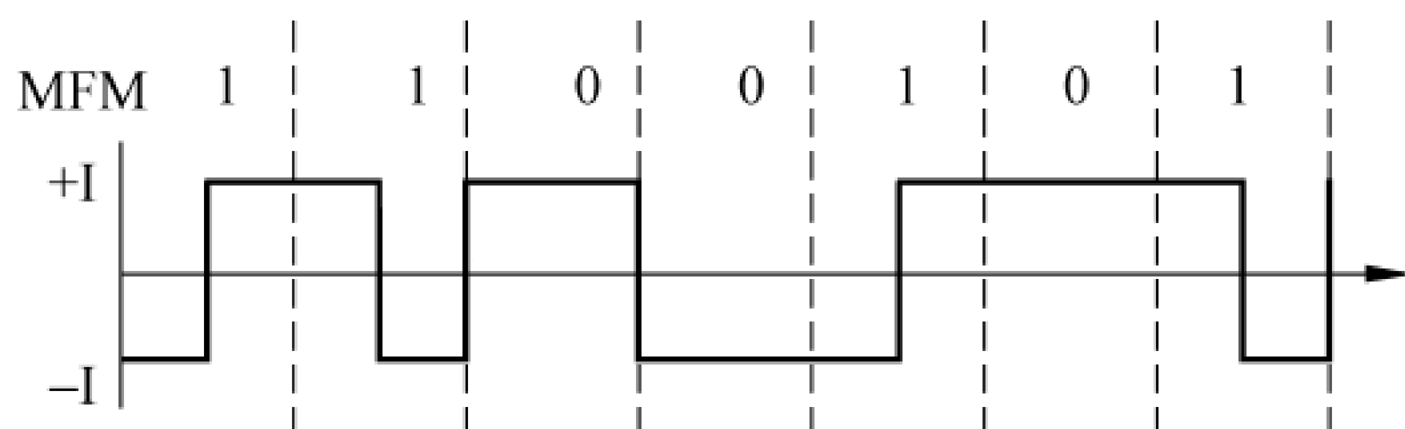


图 9.12　MFM 制写电流波形

9.3.2　硬盘存储器

硬盘存储器具有记录密度高、容量大、速度快等优点,是目前计算机存储系统中使用最普遍的一种外部存储器。

磁盘存储器有硬磁盘和软磁盘两种,软盘因携带方便、价格便宜而曾经被广泛使用,但近年来由于 U 盘的出现,使得软盘逐步被容量大、使用和携带更方便的 U 盘所淘汰。因此,以下主要介绍硬磁盘存储器、U 盘和移动硬盘、固态硬盘三类硬盘存储器。

1. 硬磁盘存储器

硬磁盘存储器的逻辑结构如图 9.13 所示。

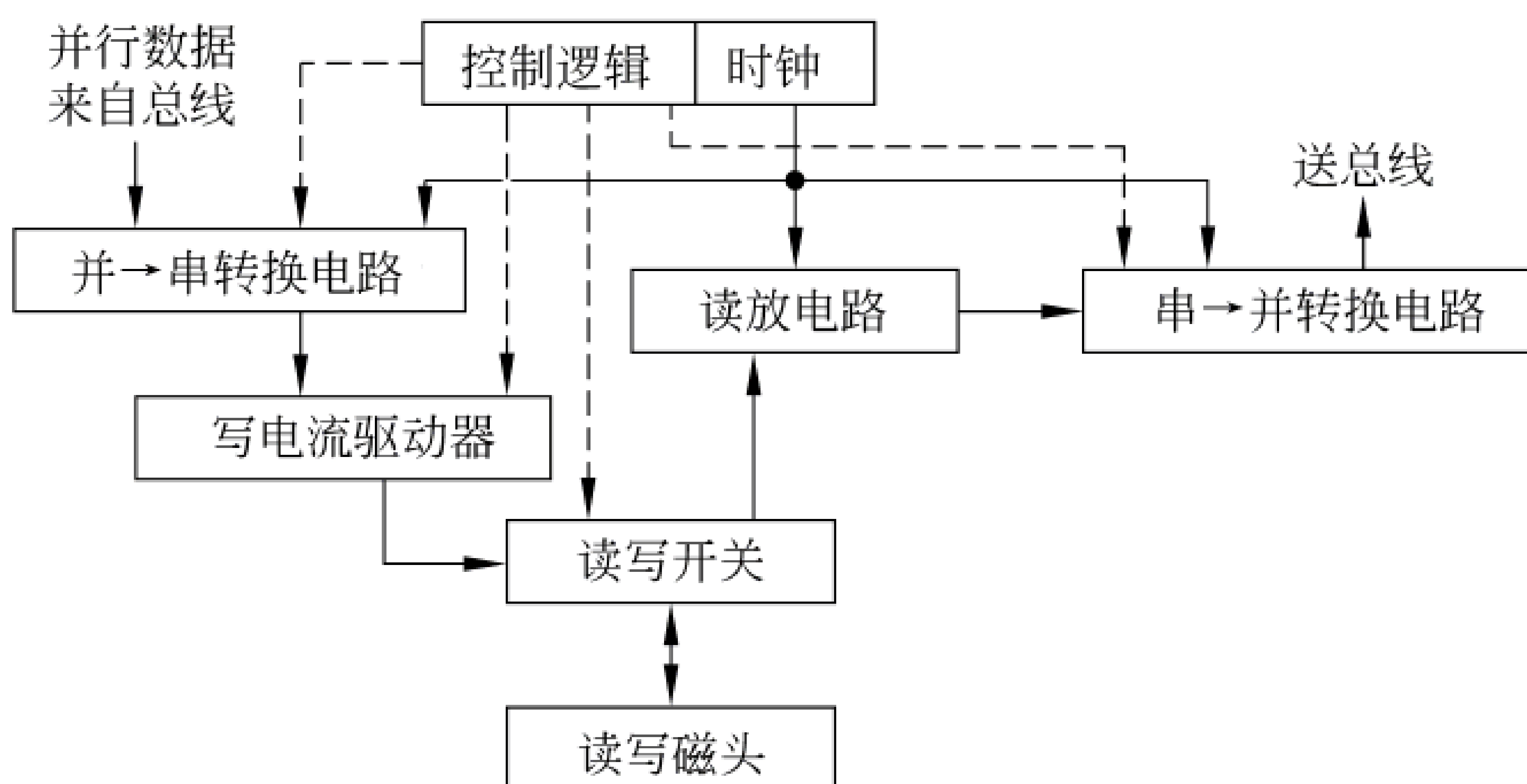


图 9.13　硬磁盘存储器逻辑结构

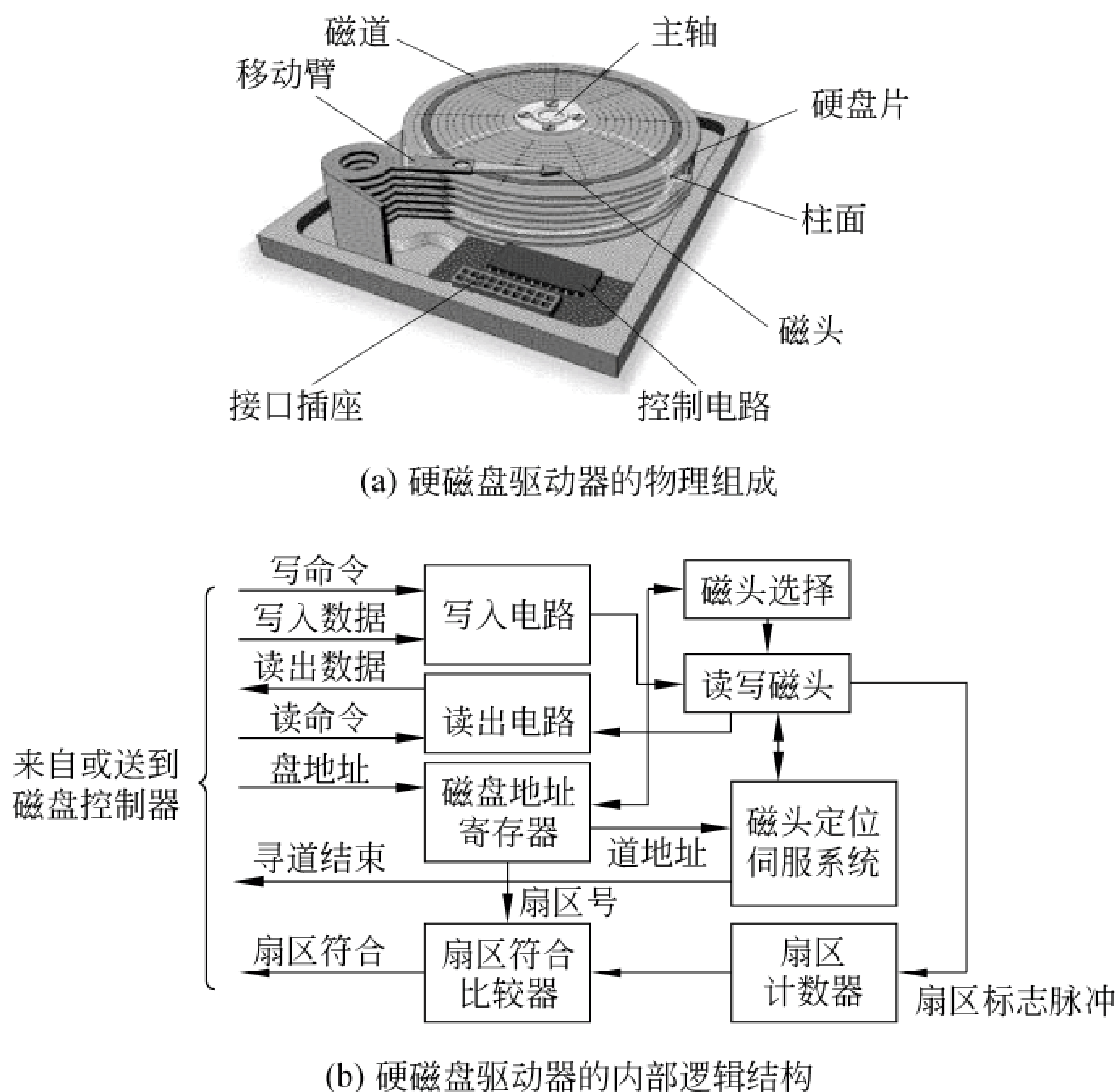
它主要由磁记录介质、磁盘驱动器、磁盘控制器三大部分组成。磁盘控制器包括控制逻辑、时序电路、"并→串"转换和"串→并"转换电路。磁盘驱动器包括读写电路、读写开关、读写磁头与磁头定位伺服系统。

(1) 硬磁盘驱动器

图 9.14 是硬磁盘驱动器的物理组成和内部逻辑结构示意图。

图 9.14(a)所示是硬磁盘驱动器的物理组成,主要由 1～5 张硬盘片、主轴、主轴电机、移动臂、磁头和控制电路等部分组成,通过接口与磁盘控制器连接,每个盘片的两个面上各

有一个磁头，因此，磁头号就是盘面号。磁头和盘片相对运动形成的圆构成一个磁道（track），磁头位于不同的半径上，则得到不同的磁道。多个盘片上相同磁道形成一个柱面（cylinder），所以，磁道号就是柱面号。信息存储在盘面的磁道上。在读写磁盘时，总是写完一个柱面上所有的磁道后，再移到下一个柱面。磁道从外向里编址，最外面的为磁道0。每个磁道又分为若干扇区（sector），按扇区为单位进行磁盘读写。



(a) 硬磁盘驱动器的物理组成

(b) 硬磁盘驱动器的内部逻辑结构

图9.14　硬磁盘驱动器的物理组成和内部逻辑结构

图9.14(b)所示是硬磁盘驱动器的内部逻辑。磁盘读写是指根据主机访问控制字中的盘地址（柱面号、磁头号、扇区号）读写目标磁道中的指定扇区。因此操作可归纳为寻道、旋转等待和读写三大类。

寻道操作：磁盘控制器把盘地址送到盘驱动器的磁盘地址寄存器后，便产生寻道命令，启动磁头定位伺服系统进行磁头定位操作。此操作完成后，发出寻道结束信号给磁盘控制器，并转入旋转等待操作。

旋转等待操作：盘片旋转时，索引标志产生的脉冲将扇区计数器清零，以后每来一个扇区标志，扇区计数加1，把计数内容与磁盘地址寄存器中的扇区地址进行比较，如果一致，则输出扇区符合信号，说明要读写的信息已经转到磁头下方。

读写操作：扇区符合信号送给控制器后，控制器的读写控制电路开始动作。如果是写操作，就将数据送到写入电路，写入电路根据记录方式生成相应的写电流脉冲；如果是读操作，则由读出放大电路读出内容送磁盘控制器。

(2) 硬磁盘控制器

硬磁盘控制器是主机与硬磁盘驱动器之间的接口。磁盘存储器是高速外设，所以磁盘控制器和主机之间采用成批数据交换方式。

主机与磁盘控制器数据交换的控制逻辑如图9.15所示。磁盘上的数据由读磁头读出

后送到读出放大器，然后进行数据与时钟的分离，再进行串/并数据转换和格式变换送到数据缓冲器，经DMA控制将数据传送到主存储器。

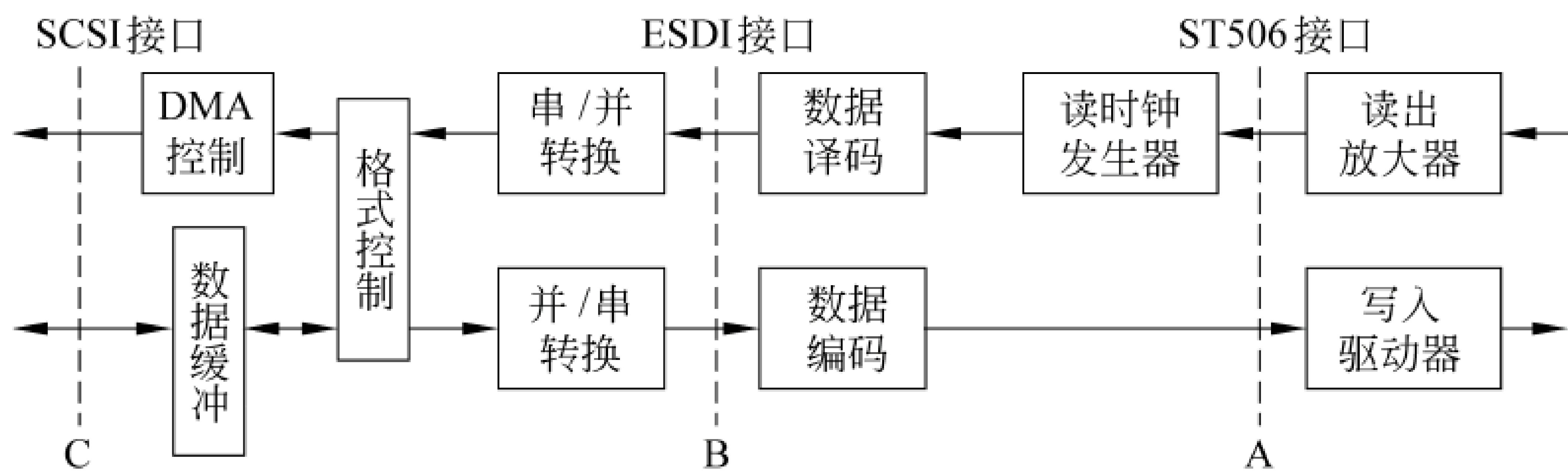


图9.15　硬磁盘控制器接口逻辑

磁盘控制器与磁盘驱动器之间并没有明确的界线，两者之间交界面的划分有几种方式。如果交界面设在图9.15的A点，驱动器只完成读写和放大，数据分离后的控制逻辑都划入磁盘控制器，例如ST506磁盘控制器就是这样。若交界面设在图9.15的B点，则在驱动器中包含数据分离电路，而磁盘控制器仅有串/并数据转换和格式变换等逻辑。例如ESDI接口属于这种形式。第三种方式的交界面设在图9.15的C点，全部控制功能均在驱动器内，主机与盘驱动器之间采用标准的通用接口，例如SCSI接口则属于这种形式。

(3) 磁盘的记录格式

数据在磁盘上的记录格式分定长记录格式和不定长记录格式两种。目前大多采用定长记录格式。图9.16是温切斯特磁盘的磁道格式示意图，它采用定长记录格式。最早的硬磁盘由IBM公司开发，称为温切斯特(Winchester，地名)盘，简称温盘，它是几乎所有现代硬盘产品的原型。

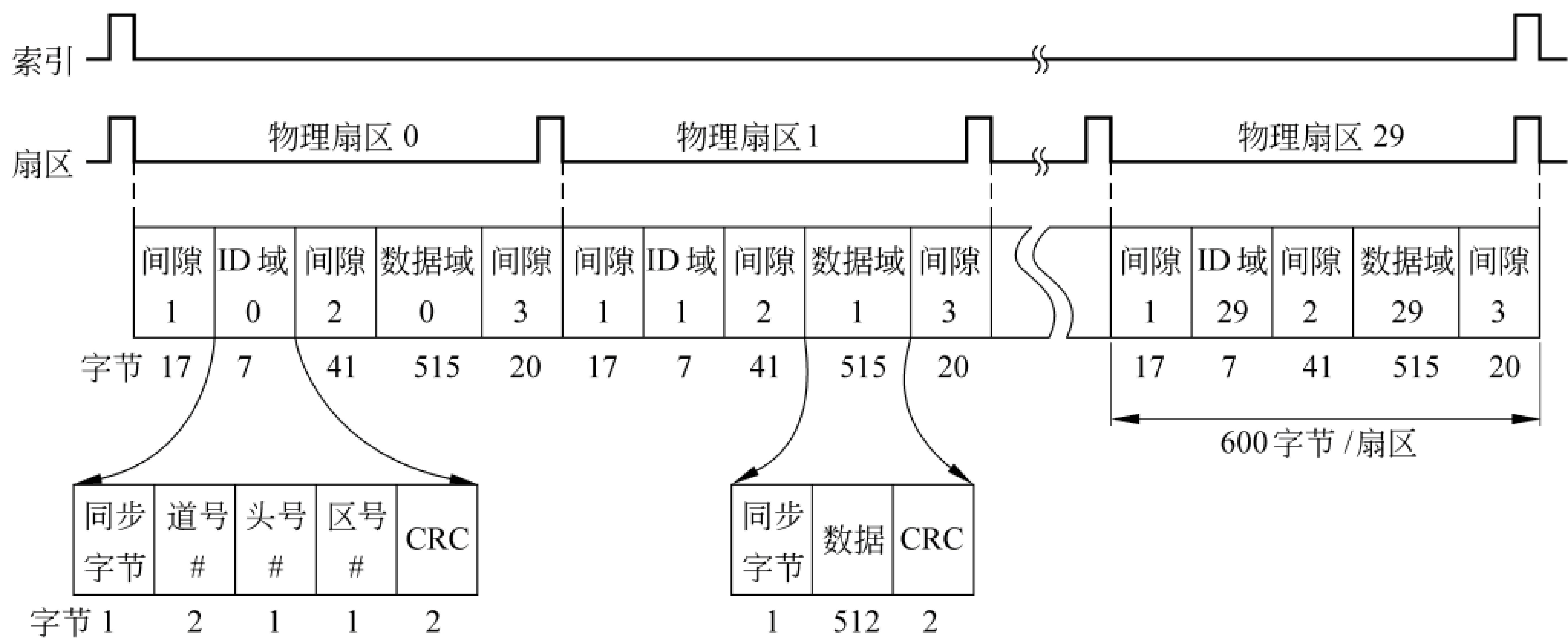


图9.16　温切斯特磁盘的磁道记录格式

每个磁道由若干个扇区(也称扇段)组成，每个扇区记录一个数据块，每个扇区由头空、ID域、间隙、数据域和尾空组成。头空占17个字节，不记录数据，用全1表示，磁盘转过该区域的时间是留给磁盘控制器作准备用的；ID域由同步字节、磁道号、磁头号、扇区号和相应的CRC码组成，同步字节标志ID域的开始；数据域占515个字节，由同步字节、数据和相应的CRC码组成，其中真正的数据区占512字节；尾空是在数据块的CRC码后的区域，占20个字节，也用全1表示。

(4) 硬磁盘的主要技术指标

硬磁盘的未格式化容量是指按道密度和位密度计算出来的容量,它包括了头空、ID域、CRC码等信息,是可利用的所有磁化单元的总数,未格式化容量(或非格式化容量)比格式化后的实际容量要大。

对于低密度存储方式,因为每个磁道的容量相等,所以,其未格式化容量的计算方法为:

磁盘总容量＝记录面数×理论柱面数×内圆周长×位密度

由于磁盘每面的有效记录区域是一个环,磁道在这个环内沿径向分布。故理论上的柱面数应该等于:(有效记录区外径－有效记录区内径)÷2×道密度。此外,对于每个磁道具有同样多信息的磁盘,其内圆磁道的记录密度最大,一个磁道能记录的二进制信息位的理论值应等于内圆周长×位密度。

格式化后的实际容量只包含数据区。通常,记录面数为盘片数的两倍,扇区大小为512B(新标准规定为4KB),所以,磁盘数据总容量(又称格式化容量)的计算公式为:

磁盘实际数据容量＝2×盘片数×磁道数/面×扇区数/磁道×512B/扇区

前面已经提到,数据传输率是单位时间内从磁盘盘面上读出或写入的二进制信息量。由于磁盘在同一时刻只有一个磁头进行读写,所以数据传输率等于单位时间内磁头划过的磁道弧长乘以位密度。即:

数据传输率＝每分钟转速÷60×内圆周长×位密度

磁盘响应读写请求的过程如下:首先将读写请求在队列中排队,出队列后由磁盘控制器解析请求命令,然后进行寻道、旋转等待和读写数据三个过程。

因此总的响应时间的计算公式为:

响应时间＝排队延迟＋控制器时间＋寻道时间＋旋转等待时间＋数据传输时间

磁盘上的信息以扇区为单位进行读写,上式中后面三个时间之和称为平均存取时间T。即:

T＝寻道时间＋旋转等待时间＋数据传输时间

寻道时间为磁头移动到指定磁道所需时间;旋转等待时间指要读写的扇区旋转到磁头下方所需要的时间;数据传输时间(transfer time)指传输一个扇区的时间(大约0.01ms/扇区)。由于磁头原有位置与要寻找的目的位置之间远近不一,故寻道时间和旋转等待时间只能取平均值。磁盘的平均寻道时间一般为5～10ms,平均等待时间取磁盘旋转一周所需时间的一半,大约4～6ms。假如磁盘转速为6000转/分,则平均等待时间约为5ms。因为数据传输时间相对于寻道时间和等待时间来说非常短,所以,磁盘平均存取时间通常近似等于平均寻道时间和平均等待时间之和。

硬磁盘驱动器与主机的接口有多种,一般文件服务器使用SCSI接口,而普通的PC前些年多使用并行ATA(即IDE)接口,目前大多使用串行ATA(即SATA)接口。

***2. 冗余磁盘阵列**

计算机发展过程中,人们比较注重处理器和主存性能的改进,而对辅存性能的改进不太注重,因而造成CPU和主存的性能提高得比辅存快,使总体性能提高不均衡。为了改善磁盘存储器的性能,1988年美国加州大学伯克利分校一个研究小组提出了一种称为RAID(Redundant Arrays of Inexpensive Disk,廉价磁盘冗余阵列)的技术,大大改善了辅存的性能。

RAID技术的基本思想是将多个独立操作的磁盘按某种方式组织成磁盘阵列，以增加容量；利用类似于主存中的多模块交叉技术，将数据存储在多个盘体上，通过使这些盘并行工作来提高数据传输速度；并用冗余磁盘技术来进行错误恢复以提高系统可靠性。

RAID技术有以下三个特性：

① RAID由一组物理磁盘驱动器组成，在操作系统下它们被视为单个逻辑驱动器。

② 数据分布在一组物理磁盘上，可以连续分布也可以交叉分布，交叉分布时可以按小条带交叉分布，也可以按大数据块交叉分布。

③ 冗余磁盘用于存储校验信息，保证磁盘万一损坏时能恢复数据。

目前已知的RAID方案分为8级(0～7级)，并由此派生出RAID 10(结合0和1级)、RAID 30(结合0和3级)和RAID 50(结合0和5级)。但这些级别不是简单地表示层次关系，而是表示具有上述三个共同特性的不同设计结构。

(1) RAID 0

RAID 0不遵循以上特性(3)，没有冗余，数据分布在多个物理磁盘上，适用于容量和速度要求高的非关键数据存储的场合，与单个大容量磁盘相比，有以下两个优点：

① 采用连续分布或大数据块交叉分布时，如果两个I/O请求访问不同盘上的数据，则两个盘可并行传送，使I/O请求同时响应，因而减少了I/O排队时间，具有较快的I/O响应能力。因而它适合要求I/O响应速度快的场合，如订票系统等。

② 采用小条带交叉分布时，一个I/O请求很可能占多个条带，并分布在不同的盘上，因而可并行传送同一个I/O请求的不同条带上的数据块，所以数据传输率较高。因而，适用于大容量I/O请求的场合，如流媒体播放系统、图像处理、CAD系统等。

(2) RAID 1

RAID 1采用镜像盘实现一对一冗余，有以下三个特点：

① 一个读请求可由其中一个定位时间更少的磁盘提供数据。

② 一个写请求对两个磁盘中相应信息并行更新，故写性能由两次中较慢的一次决定。

③ 数据恢复简单，因为当一个磁盘损坏时，数据仍能从另一个磁盘中读取。

RAID 1的可靠性高，但价格昂贵。常用于可靠性要求很高的场合。

(3) RAID 2

RAID 2用海明校验生成多个冗余校验盘，实现纠正一位错误、检测两位错误的功能。采用小条带交叉分布方式(有时一个条带为一个字或一个字节)，这样，可获得较高数据传输率，但I/O响应时间较长。因为采用海明码，所以校验盘与数据盘成正比，因而冗余信息开销太大，价格较贵，所以RAID 2已不再被使用。

(4) RAID 3

RAID 3采用奇偶校验生成单个冗余盘，与RAID 2相同，也采用小条带交叉分布方式，数据传输率高，但I/O响应时间较长。

RAID中损坏的数据可以通过其他磁盘重新生成。RAID 3采用以下数据恢复操作。假定考虑一个有5个磁盘的阵列，其中$X0$～$X3$为数据盘，P是奇偶校验盘，奇偶校验的第i位计算公式如下：

$$P(i)=X3(i)\oplus X2(i)\oplus X1(i)\oplus X0(i)$$

若磁盘$X0$损坏，则在上述等式两边同时异或$P(i)\oplus X0(i)$，得到以下等式：

$$X0(i)=P(i)\oplus X3(i)\oplus X2(i)\oplus X1(i)$$

因此,某数据磁盘中的任一数据条带的内容都能从剩余磁盘的相应条带中重新生成。

(5) RAID 4

与 RAID 3 一样,RAID 4 也采用一个冗余盘存放奇偶校验位,不过 RAID 4 采用的是大数据块交叉方式,每个磁盘的操作独立进行,所以多个小数据量的操作可以在多个磁盘上并行进行,以同时响应多个 I/O 请求,具有较快的 I/O 响应时间,可用于银行、证券等事务处理系统。

RAID 4 的数据恢复方式与 RAID 3 相同,比较容易。但对于写操作,因为每次写都要对校验盘进行相应的校验数据更新,因而校验盘成为 I/O 瓶颈。在"大量写"(涉及所有磁盘写)时,因为奇偶校验位可全部用新数据计算得到,所以,无须读出原数据,计算出校验位后,直接写入奇偶校验盘和数据盘即可;而在"少量写"(只涉及个别磁盘写)时,因为一次写操作包含两次读和两次写,所以有"写损失"。因为 RAID 4 采用的是大数据块交叉方式,所以,通常发生的是"少量写"。例如,假定考虑一个有 5 个磁盘的阵列,其中 $X0\sim X3$ 为数据盘,P 是奇偶校验盘,初始时每位 i 有下列关系式:

$$P(i)=X3(i)\oplus X2(i)\oplus X1(i)\oplus X0(i)$$

假定仅将 $X0(i)$ 更新为 $X'0(i)$,而 $X3(i)$、$X2(i)$、$X1(i)$ 不变,则新的校验位 $P'(i)$ 如下:

$$\begin{aligned}P'(i)&=X3(i)\oplus X2(i)\oplus X1(i)\oplus X'0(i)\\&=X3(i)\oplus X2(i)\oplus X1(i)\oplus X'0(i)\oplus X0(i)\oplus X0(i)\\&=P(i)\oplus X0(i)\oplus X'0(i)\end{aligned}$$

由此可见,要更新一个 $X0(i)$,必须先读 $P(i)$ 和 $X0(i)$,然后写 $X'0(i)$ 和 $P'(i)$。

(6) RAID 5

RAID 5 与 RAID 4 的组织方式类似,只是奇偶校验块分布在各个磁盘中,因此,所有磁盘地位等价,这样可提高容错性,并且避免了使用专门校验盘时潜在的 I/O 瓶颈。因为它与 RAID 4 一样,也采用独立存取技术和大数据块交叉分布方式,所以 I/O 请求的响应速度快。由此可见,RAID 5 成本不高但效率高,因而被广泛使用。

(7) RAID 6

如图 9.17 所示,RAID 6 与 RAID 4 和 RAID 5 一样,采用独立存取技术和大数据块交叉分布方式。所不同的是 RAID 6 的冗余信息分布在所有磁盘上,且采用双维块奇偶校验(P0 代表第 0 条块的校验值,而 PA 代表数据块 A 的校验值),因而 RAID 6 容许双盘出错。因此,可用于要求数据绝对不能出错的场合。

块 A_0	块 B_0	块 C_0	块 P_0
块 A_1	块 B_1	块 P_1	块 P_A
块 A_2	块 P_2	块 P_B	块 D_0
块 P_3	块 P_C	块 C_1	块 D_1
块 P_D	块 B_2	块 C_2	块 D_2

图 9.17　RAID 6 的双维校验块分布

由于引入了两个奇偶校验值，因而控制器的设计变得十分复杂，写入速度也比较慢，用于计算奇偶校验值和验证数据正确性所花费的时间比较多。由此可见，RAID 6 以增大时间开销为代价保证了高度可靠性。

(8) RAID 7

RAID 7 是带 cache 的磁盘阵列，它在 RAID 6 的基础上，采用 cache 技术使传输率和响应速度都有较大提高，cache 分块大小和磁盘阵列中数据分块大小相同，一一对应。有两个独立的 cache，双工运行。在写入时将数据同时分别写入两个独立的 cache，这样，即使其中有一个 cache 出故障，数据也不会丢失。写入磁盘阵列以前，先写入 cache 中，然后，同一磁道的信息在一次操作中完成；读出时，先从 cache 中读出，cache 中没有要读的信息时，才从 RAID 中读。RAID 7 将 cache 和 RAID 技术结合，弥补了 RAID 的不足，从而将高效、快速、大容量、高可靠性以及灵活方便的存储系统提供给用户。

***3. U 盘和移动硬盘**

随着计算机技术与应用的发展，近年来开始普遍使用 U 盘和移动硬盘进行信息存储和交换。

U 盘也称为闪存盘，与上述硬磁盘不同，它不是磁表面存储器，而是采用 flash 存储器(即闪存)做成，属于非易失性半导体存储器。闪存沿用了 EPROM 的简单结构和浮栅/热电子注入的编程写入方式，又兼备 E^2PROM 的可擦除特点，可在计算机内进行擦除和编程写入。因此又称为快擦型电可擦除重编程 ROM。U 盘体积小、重量轻、容量比软盘和光盘大的多，而且可以具有保护功能，使用寿命可长达数年之久。而且，利用 USB 接口，可以与几乎所有计算机连接。

比 U 盘容量更大的用作计算机系统数据备份的移动设备是移动硬盘。它是由微型硬盘配上特制的硬盘盒构成的一个大容量存储器。通过 USB 和 IEEE 1394 接口和计算机连接，可以随时插拔。其优点是容量大、兼容性好、速度快、体积小、重量轻、携带方便、安全可靠。

***4. 固态硬盘**

近年来一种称为固态硬盘(Solid State Disk，SSD)的新产品开始在市场上出现，也被称为电子硬盘。这种硬盘并不是一种磁表面存储器，而是一种使用 NAND 闪存组成的外部存储系统，和 U 盘并没有本质差别，只是容量更大，存取性能更好。它用闪存颗粒代替了磁盘作为存储介质，利用闪存的特点，以区块写入和抹除的方式进行数据的读取和写入。电信号的控制使得固态硬盘的内部传输速率远远高于常规硬盘。有测试显示，使用固态硬盘以后，Windows 的开机速度可以被提升至 20 秒以内，这是基于常规硬盘的计算机系统难以达到的速度性能。

与常规硬盘相比，除速度性能外，固态硬盘还具有抗震动好、安全性高、无噪音、能耗低、发热量低和适应性高的特点。由于不需要电机、盘片、磁头等机械部分，固态硬盘工作过程中没有任何机械运动和震动，因而抗震性好，使数据安全性成倍提高，并且没有常规硬盘的噪音；由于不需要马达工作，固态硬盘的能耗也得到了成倍的降低，只有传统硬盘的 1/3 甚至更低，延长了靠电池供电的设备的连续运转时间；且由于没有电机等机械部件，其发热量大幅降低，延长了其他配件的使用寿命。此外，固态硬盘的工作温度范围很宽(－40℃～85℃)，因此，其适应性上也远高于常规硬盘。

固态硬盘在刚出现时与最高速的常规硬盘相比在读写性能方面各有上下，而且价格也较高。但随着相应技术的不断发展，目前固态硬盘的读写性能基本上超越了常规硬盘，且价格也不断下降。由于固态硬盘具有以上优点，加上其今后的发展潜力比传统硬盘要大得多，因而固态硬盘有望逐步取代传统硬盘。

固态硬盘的接口规范和定义、功能及使用方法与传统硬盘完全相同，在产品外形和尺寸上也与普通硬盘一致。目前主要有 2.5 英寸与 1.8 英寸两种尺寸；接口标准上使用 SATA 和 IDE 接口；目前市场上产品容量在 32～256GB 之间，但已发表的最大容量已达 1TB。在访问性能方面，目前固态硬盘的平均访问时间大约是常规硬盘的 1.5 倍，写入速度可达常规硬盘的 1.5 倍，读取速度可达常规硬盘的 2～3 倍；而 CPU 占用率可比常规硬盘下降 5～10 倍。

固态硬盘目前主要的问题是使用寿命和价格。由于闪存的擦写次数有限，所以频繁擦写会降低其写入使用寿命；而价格上目前固态硬盘的价格远远高于常规硬盘。但随着技术和生产工艺的不断进步，固态硬盘的写入使用寿命会不断提高，且价格也将不断下降。

*9.3.3 磁带存储器

磁带存储器可用作海量数据备份存储设备。磁带有许多种，按带宽分有 1/4 英寸和 1/2 英寸；按带长分有 2400 英尺、1200 英尺和 600 英尺等；按外形分有开盘式和盒式磁带；按记录密度分有 800 位/英寸、1600 位/英寸、6250 位/英寸等；按带面并行的磁道数分有 9 道、16 道等。计算机系统中一般采用 1/2 英寸开盘式磁带和 1/4 英寸盒式磁带。

磁带机也有许多种，按走带速度分有高速、中速和低速磁带机。按磁带机规模分，有标准 1/2 英寸磁带机、盘式磁带机和海量宽磁带机三种。按磁带的记录格式分有启停式和数据流式。

为了寻找记录区，必须驱动磁带正走或反走，读写完毕后使磁头停在两个记录区之间。因此要求磁带机在结构上和电路上采取相应措施，以保证磁带以一定的速度平稳地运动和快速启停。下面简要说明两种磁带机的结构。

1. 开盘式启停磁带机

开盘式磁带缠绕在圆形带盘上，磁带首端可以取出，磁带上的信息以数据块为单位，在数据块之间，磁带机需要启动和停止。开盘式启停磁带机主要由走带机构、磁带缓冲机构、带盘驱动机构和磁头等部分组成。走带机构的作用是带动磁带运动；磁带缓冲机构的作用是减小磁带运动中的惯性，以便快速启停；带盘驱动机构的作用是控制带盘电机的方向和速度，使放带盘和收带盘都能正转和反转，并且调节放带盘和收带盘的转速；磁头和磁盘机的磁头原理相同，但是磁带机将多个磁头组装在一起，构成组合磁头，以便同时对各道进行读写。

1/2 英寸 9 道启停式磁带是一种国际通用的标准磁带，其记录格式见图 9.18。

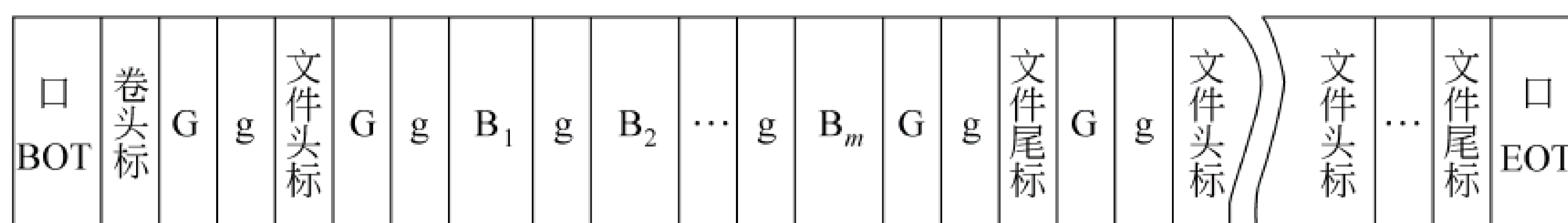


图 9.18　1/2 英寸 9 道启停式磁带记录格式

每盘磁带都有始端标记 BOT 和末端标记 EOT。标记用反光的金属薄膜制成，光电检测元件可检测出这两个标记并产生相应的电信号。信息可用两种形式存储：一种是文件形式，一盘磁带可记录若干个文件，一个文件又分若干个数据块(B_1，B_2，…，B_m)。每个文件的始末有文件头标和文件尾标。卷头标、索引、文件头标和文件尾标均为 80B，其内容随操作系统而不同。文件头标和文件尾标后有 3.75 英寸的间隙 G，数据块间有 0.6 英寸的间隙 g。另一种存储形式是数据块形式，磁带可在数据块之间启停，在 9 道带中，8 个道是数据磁道，另一道是这 8 位数据的奇偶校验位。在每个数据块内部，每条磁道沿走带方向还有 CRC 校验码。

2. 数据流磁带机

数据流磁带机将数据连续地写在磁带上，在数据块之间插入记录间隙，这样磁带机在数据块之间不用启停。此外，它采用电子控制代替机械控制，简化了磁带机的机械结构，降低成本，提高了可靠性。数据流磁带机不是多位并行读写，而是和磁盘一样采用逐道串行读写方式，记录信息从 0 磁道开始，偶数道从 BOT 到 EOT，而奇数道从 EOT 到 BOT，依次首尾相接。因而它的记录格式与启停式磁带机不同。

数据流磁带机有 1/2 英寸开盘式和 1/4 英寸盒式两种。1/4 英寸盒式数据流磁带也是通用的标准磁带，其中 9 磁道记录格式见图 9.19，它包括前同步、数据块标志(1B)、用户数据(512B)、地址(4B)、CRC 校验码(2B)和后同步。

前同步	1B 数据块标志	512B 用户数据	4B 地址	2B CRC	后同步

图 9.19　1/4 英寸 9 道数据流磁带记录格式

*9.3.4　光盘存储器

光盘存储器是一种采用聚焦激光束在盘形介质上高密度地记录信息的存储装置。具有记录密度高、存储容量大、信息保存寿命长、工作稳定可靠、环境要求低等特点，现已广泛应用于存储各种数字化信息，包括大型数据处理系统和办公自动化系统中的文件、声音和图像的存档与检索等领域，是磁盘机的重要后援设备。

1. 光盘存储器的基本工作原理

(1) 只读型光盘的信息存储机理

制作时先利用激光束，将经过编码和调制的数据信息螺旋状刻录在玻璃母盘上，后经显影、腐蚀，使曝光部分达到所需的深度，得到一张阳母盘；再在阳母盘上镀上一层厚的金属层，使它和阳盘分离，得到一张金属母盘(阴盘)，利用它来制作压膜，把压膜压在加热的聚碳酸酯盘上，后者的表面就会出现与金属母盘成镜像对称的小凹坑。这些凹坑就是录制的数据，凹坑的边缘处表示“1”，而凹坑内和凹坑外的平坦部分表示“0”。这种过程与集成电路生产中使用的微缩照相技术非常相似，所不同的是，要产生宽度大约 0.6μm 的凹坑所需的光点不仅尺寸要小，而且强度要高，为此必须采用激光作为光源，并采用良好的光学系统才能实现。

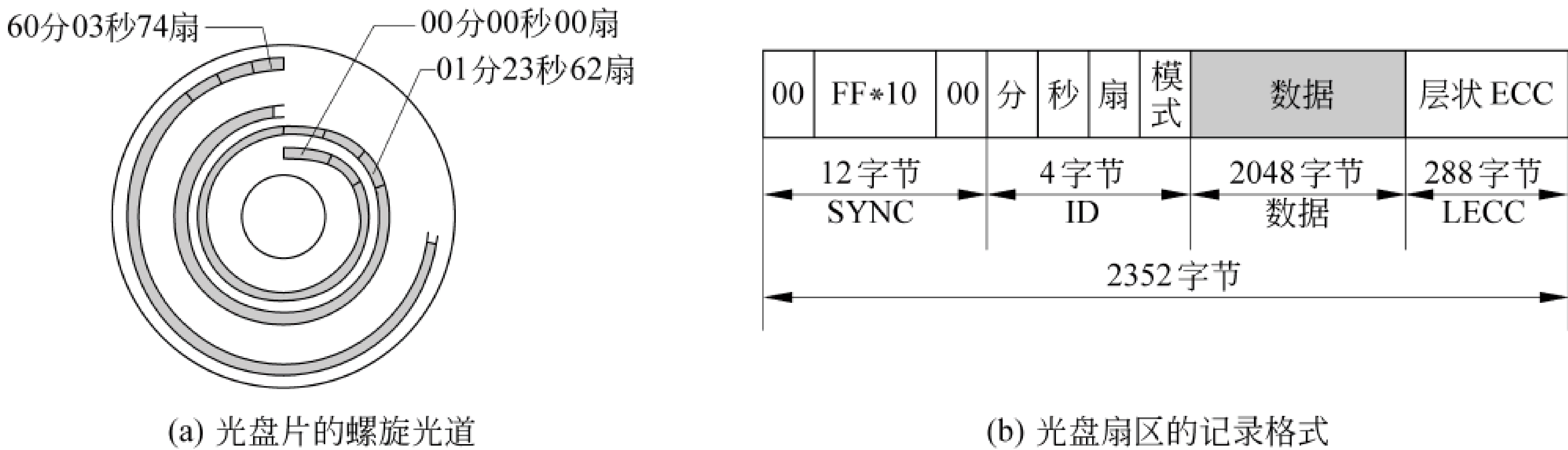
(2) 一次写入型光盘的信息存储机理

本质上讲，一次写入型光盘的信息存储机理与前述只读型光盘一样。不同之处在于它增加了一层有机染料作为记录层，其调制激光束输出强度也要高得多，当写入激光束聚焦

后，照在记录层上时，有机染料被加热后有一部分熔解掉，形成代表信息的凹坑。读出数据时，在高温激光束的照射下，熔解处与未熔解处的颜色和亮度不同，因而可以区分读出的是0还是1。

2. 光盘上信息的记录格式

光盘上用于记录数据的是一条由里向外的连续的螺旋状光道。如图9.20(a)所示，信息按照一个个扇区串行记录在光道上，扇区是光盘最小可寻址单位，按时间方式来定位，扇区地址为分、秒、扇区号。由于光盘的基本恒定线速度为每秒钟读75个扇区，所以每秒钟内扇区的编号为0～74。图9.20(b)给出了CD-ROM光盘上一个扇区的记录格式。每个扇区共2352个字节，包括12字节的同步字段，4个字节的ID字段，2048个字节的数据字段和288个字节的校验码字段。



(a) 光盘片的螺旋光道　　(b) 光盘扇区的记录格式

图9.20　光盘上数据的存放形式

3. 光盘存储器的组成

光盘存储器由光盘片和光盘驱动器两个部分组成。

(1) 光盘片的分类

光盘的基片是一种耐热的有机玻璃。光盘存储数据的原理与磁盘不同，它通过在盘面上压制凹坑来记录信息。

光盘片按所用激光类型来分，目前有CD盘片、DVD盘片和蓝光(BD)盘片三种。CD盘片与DVD盘片大小相同，但所用激光波长(各为780nm/650nm)、光斑直径(各为1.74μm/1.08μm)、道间距(各为1.6μm/0.74μm)和凹坑宽度(各为0.6μm/0.4μm)和凹坑最小长度(各为0.83μm/0.4μm)都不一样，因而DVD盘片的光道长度比CD盘片的长，约为11.8km，而CD盘约为5.3km。而且DVD盘有单层单面、单层双面、双层单面、双层双面4种类型，因而，容量比CD盘片高得多，CD盘片容量为650MB，而单层单面DVD容量达4.7GB。BD盘片是目前最先进的大容量光盘片，它利用波长更短(450nm)的蓝光激光进行信息读写，单层盘片的容量为25GB，读写速度可达4.5～9MBps。

光盘片按读写类型来分，可分为只读型、一次写入型和可擦写型三种。

只读型光盘由生产厂家预先用激光在盘片上蚀刻而成，信息不能改写。例如激光视盘(LV)、数码唱盘(CD-DA)、计算机用的CD-ROM、DVD-ROM、BD-ROM等均属于此类，目前这类光盘的应用已相当普及。

一次写入型光盘是指由用户一次写入、可多次读出但不能擦除的光盘。要修改的数据只能追记在盘片上的空白处，故又称为追记型光盘，它用于不需要修改的应用场合。此类光盘片有CD-R、DVD-R、DVD+R、BD-R等。

可擦写型光盘是可读可写型光盘，可以多次改写，擦写次数可达几百次甚至上千次之多。此类光盘片有CD-RW、DVD-RW、DVD+RW、BD-RW等。

（2）光盘驱动器的分类

光盘驱动器（简称光驱）由主轴驱动机构、定位机构、光头及有关的控制和驱动电路组成，光盘安装在主轴驱动机构的主轴上，由主轴电机直接驱动不断旋转。光头安装在直线电机带动的长行程定位机构上，可以对光盘上的任意光道进行信息存取。

光驱按其信息读写能力分为只读光驱和光盘刻录机两大类型；按其可读写的光盘片类型来分，可分为CD只读光驱和CD刻录机、DVD只读光驱和DVD刻录机以及DVD只读光驱和CD刻录机相结合的组合光驱（称为“Combo”光驱），此外，还有最新的一种大容量蓝色激光光驱，也分蓝光光驱（BD-ROM光驱）和蓝光刻录机（BD刻录机）两种。

9.4 I/O 接口

外部设备种类繁多，且具有不同的工作特性，因而它们在工作方式、数据格式和工作速度等方面存在很大差异。此外，由于CPU、内存等计算机主机部件采用高速元器件，使得它们和外设之间在技术特性上有很大的差异，它们各有自己的时钟和独立的时序控制，两者之间采用完全的异步工作方式。为此，在各个外设和主机之间必须要有相应的逻辑部件来解决它们之间的同步与协调、工作速度的匹配和数据格式的转换等问题，该逻辑部件就是I/O接口（I/O模块）。从功能上来说，微机中各种I/O控制器或设备控制器（包括适配器或适配卡）都是I/O接口，而在有些大型机中的I/O模块就是担负大量复杂的外设控制任务的通道或I/O处理器。

9.4.1 I/O 接口的功能

I/O接口是连接外设和主机的一个“桥梁”，因此它在外设侧和主机侧各有一个接口。通常把它在主机侧的接口称为内部接口，在外设侧的接口称为外部接口。内部接口通过系统总线和内存、CPU相连，而外部接口则通过各种接口电缆（如USB线、IEEE 1394线、串行电缆、并行电缆、网线或SCSI电缆等）将其连到外设上。因此，通过I/O接口，可以在CPU、主存和外设之间建立一个高效的信息传输“通路”。I/O接口的职能可概括为以下几个方面。

（1）数据缓冲：由于主存和CPU寄存器的存取速度非常快，而外设速度则较低，所以在I/O接口中引入数据缓冲寄存器，以达到主机和外设工作速度的匹配。

（2）错误或状态检测：在I/O接口中提供状态寄存器，以保存各种状态信息，供CPU查用。例如，设备是否完成打印或显示；是否已准备好输入数据以供主机来读取；是否发生缺纸等某种出错情况等等。接口和外设发生的出错情况有两类：一类是设备电路故障或异常情况；另一类是数据传输错，这种错误是通过在每个字符上采用一个校验码来检测的。

（3）控制和定时：提供控制和定时逻辑，以接受从系统总线来的控制和定时信号。CPU根据程序中的I/O请求，选择相应的设备进行通信，要求一些内部资源（如主存或寄存器、系统总线等）参与到I/O过程中，这样I/O接口就必须提供定时和控制功能，以协调内部资源与外设间动作的先后关系，控制数据通信过程。

(4) 数据格式转换。提供数据格式转换部件(如进行串-并转换的移位寄存器),使通过外部接口得到的数据转换为内部接口需要的格式,或在相反的方向进行数据格式转换。

(5) 与主机和设备通信。上述4个功能都必须通过I/O接口与主机或I/O接口与设备间的通信来完成。

I/O接口与主机侧进行的通信控制包括:

① 对主机通过系统总线送来的地址信息进行译码,以确定是否选中本设备;

② 接受系统总线送来的控制信息,以确定数据传送的方向等;

③ 将接口中数据缓冲寄存器或状态寄存器的信息送到系统总线,或接收系统总线送来的数据或命令信息,将其送到接口的数据缓冲寄存器或控制寄存器。

I/O接口与设备侧进行的通信控制包括:

① 将控制寄存器中的命令译码,输出到外部接口的控制线上;

② 将数据缓冲寄存器的数据发送到外部接口的数据线上;

③ 接受外设的状态或数据信息,送到接口中的状态寄存器或数据缓冲寄存器中。

数据在外设和主机之间进行传送的过程如图9.21所示。

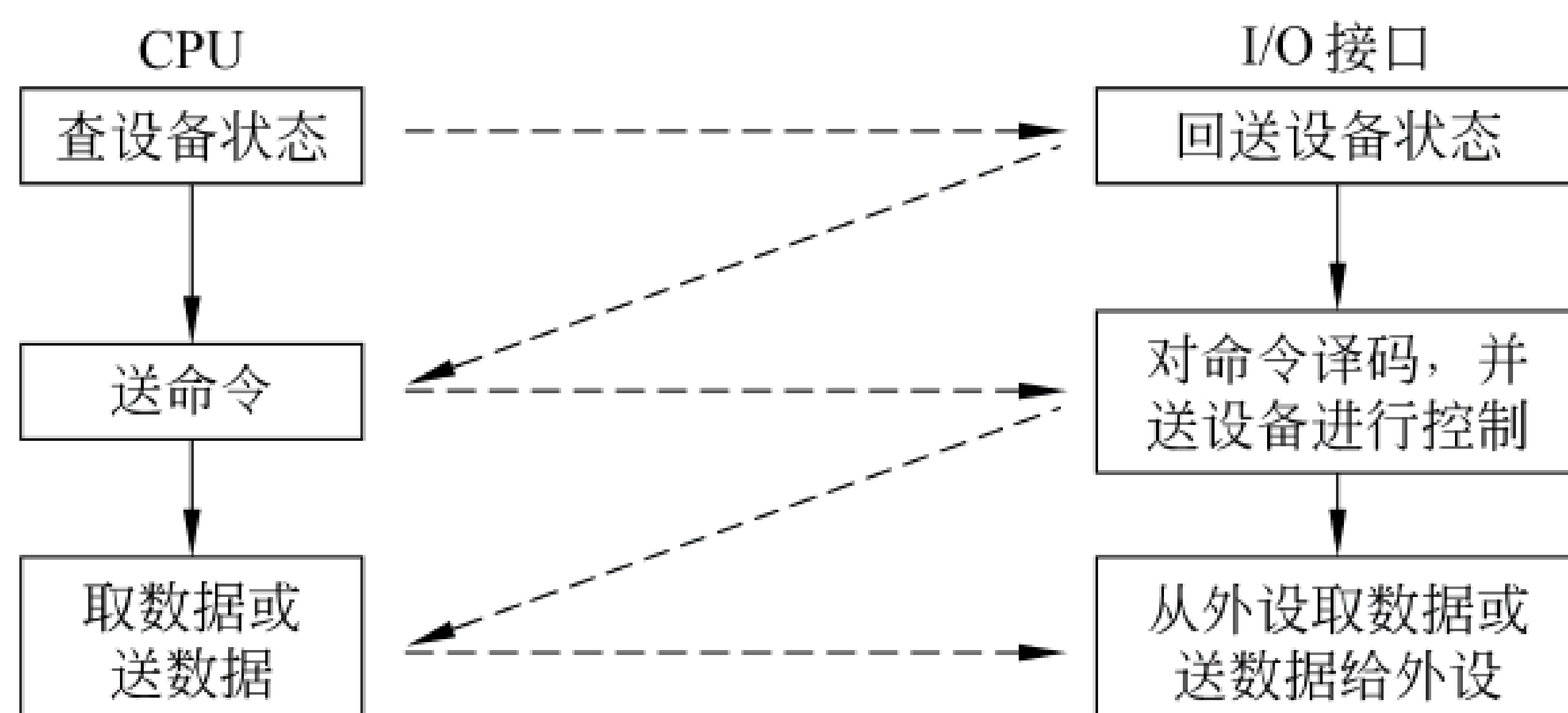


图9.21 数据在外设和主机间的传送过程

CPU通过系统总线对I/O接口进行访问和控制,读取设备的状态以了解接口和设备的情况,根据读到的状态信息向设备发相应的控制命令;在适当的时间从数据总线取数据或发送数据到数据总线。在对I/O接口取状态、发控制命令或读写数据时,都必须对CPU送到系统总线上的地址进行译码,选中对应的I/O接口才能实现对其进行访问的操作。

在外设和主机间的数据交换过程中,I/O接口起着中间桥梁的作用。在CPU的控制下,I/O接口所进行的操作包括:从外设取状态,将外设的状态和接口本身的状态回送CPU;将CPU送来的命令存入命令寄存器,并对其进行译码,然后将译码得到的控制信号送外设;从外设取数据或发送数据到外设等。

9.4.2 I/O接口的通用结构

不同的I/O接口在复杂性和控制外设的数量上相差很大,不可能一一列举,在此仅考察I/O接口的一般结构,即具有共性的部分,图9.22给出了它的通用结构。

如图9.22所示,I/O模块在主机侧通过属于系统总线(如PCI总线、扩充E/ISA总线等I/O总线)的一组信号线与内存、CPU相连。通过其中的数据线,在数据缓冲寄存器与内存或CPU的寄存器之间进行数据传送。同时接口和设备的状态信息被记录在状态寄存器中,通过数据线将状态信息送到CPU,以供查用。CPU对外设的控制信息也是通过数据线

传送，一般将其送到I/O接口的控制寄存器。从功能上来说，状态寄存器和控制寄存器在传送方向上是相反的，而且CPU对它们的访问在时间上一般是错开的，因此有的I/O接口中将它们合二为一。系统总线的地址线给出了要访问的I/O接口中寄存器的地址，它和控制信息一起被送到I/O接口中的控制逻辑部件中，其中地址译码逻辑用以选择和主机交换数据的寄存器。控制线传送来的控制信号也有可能参与地址译码，例如，可以用读写信号确定是接收寄存器还是发送寄存器。此外，控制线中还有一些仲裁信号和握手信号等也可被I/O接口使用。I/O接口中的控制逻辑部件还要能对控制寄存器中的命令字进行译码，并将译码得到的控制信号送外设，同时将数据缓冲寄存器的数据发送到外设或从外设接收数据到数据缓冲寄存器。另外，还要具有收集外设状态到状态寄存器的功能。

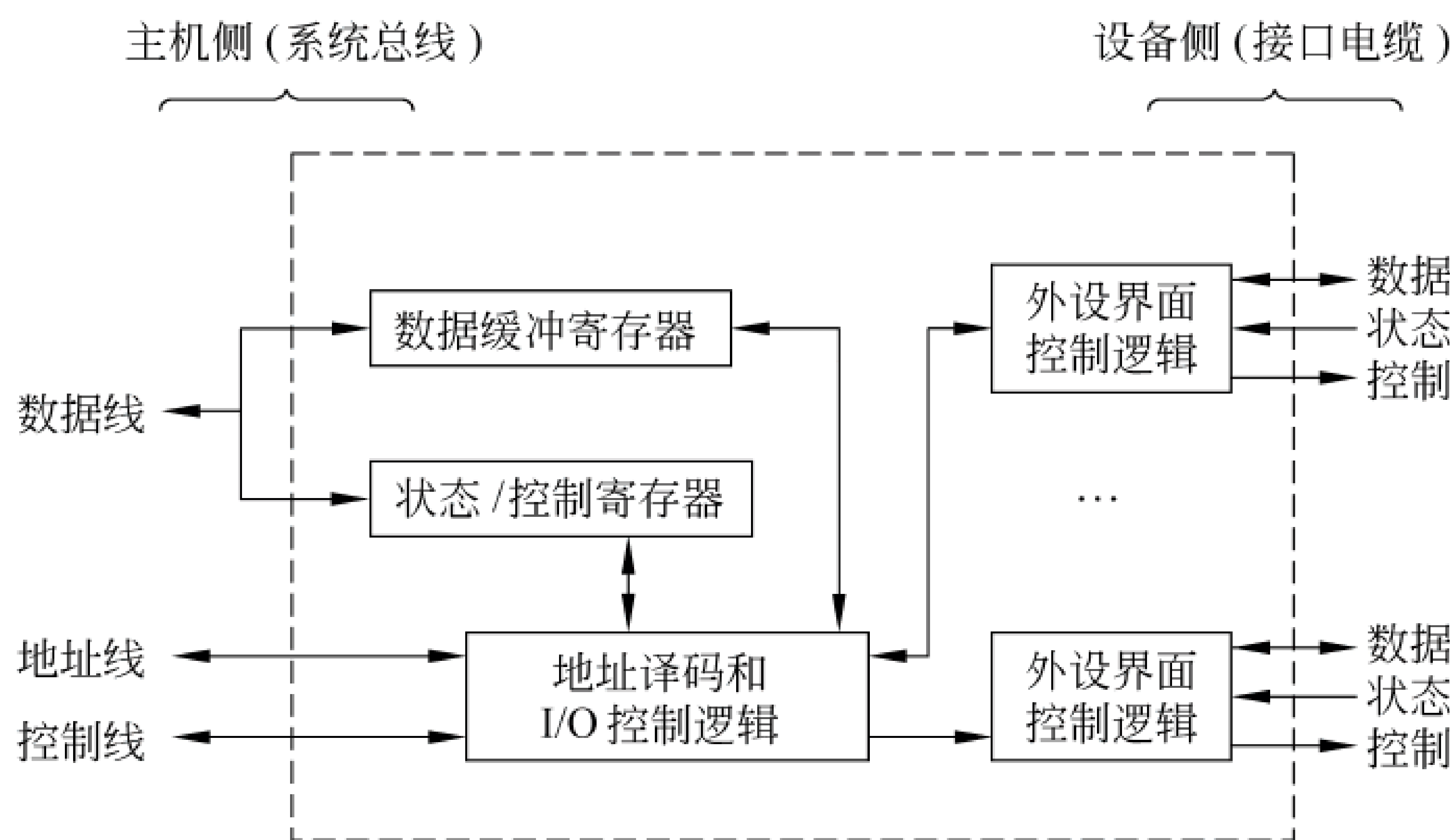


图 9.22 I/O接口的通用结构

如果某个I/O接口能够作为数据通信的主控设备(例如DMA控制器)，那么它就可以发起总线事务并控制总线进行数据传送，控制线上的控制信号由它确定，它的地址线方向是输出。

如图9.23是PC中主机和外部设备的连接关系示意图。可以看出主机和外设之间的通路为CPU和内存—I/O总线—I/O控制器—I/O接口电缆—外设。其中I/O控制器和对应的连接器插座合在一起称为I/O接口或I/O模块。键盘、鼠标、打印机、磁盘等的I/O控制器比较简单，基本上都集成在PC主板芯片中。音/视频、显示器、网络等I/O控制器比较复杂且规格繁多，这些设备在早期常制作成扩充卡(也称为适配卡或控制卡)插在PC主板的I/O总线槽中。随着集成电路技术的发展，芯片组的集成度越来越高，因而，越来越多的I/O控制器(如声卡、网卡等)已集成在芯片组中。

*9.4.3 操作系统对I/O的支持

I/O接口的引入给外设与主机进行信息交换提供了有效“通路”。但是还必须提供一种手段，让CPU能方便地找到要进行信息交换的设备，并将用户的I/O请求转换成对设备的控制命令。

现代计算机I/O系统的复杂性一般都隐藏在操作系统中，最终用户或用户程序只需通过一些简单的命令或系统调用就能使用各种外设，而无须了解设备具体工作细节。对于最终用户，操作系统通过命令行方式、批命令方式或图形界面方式为其提供直接使用计算机资

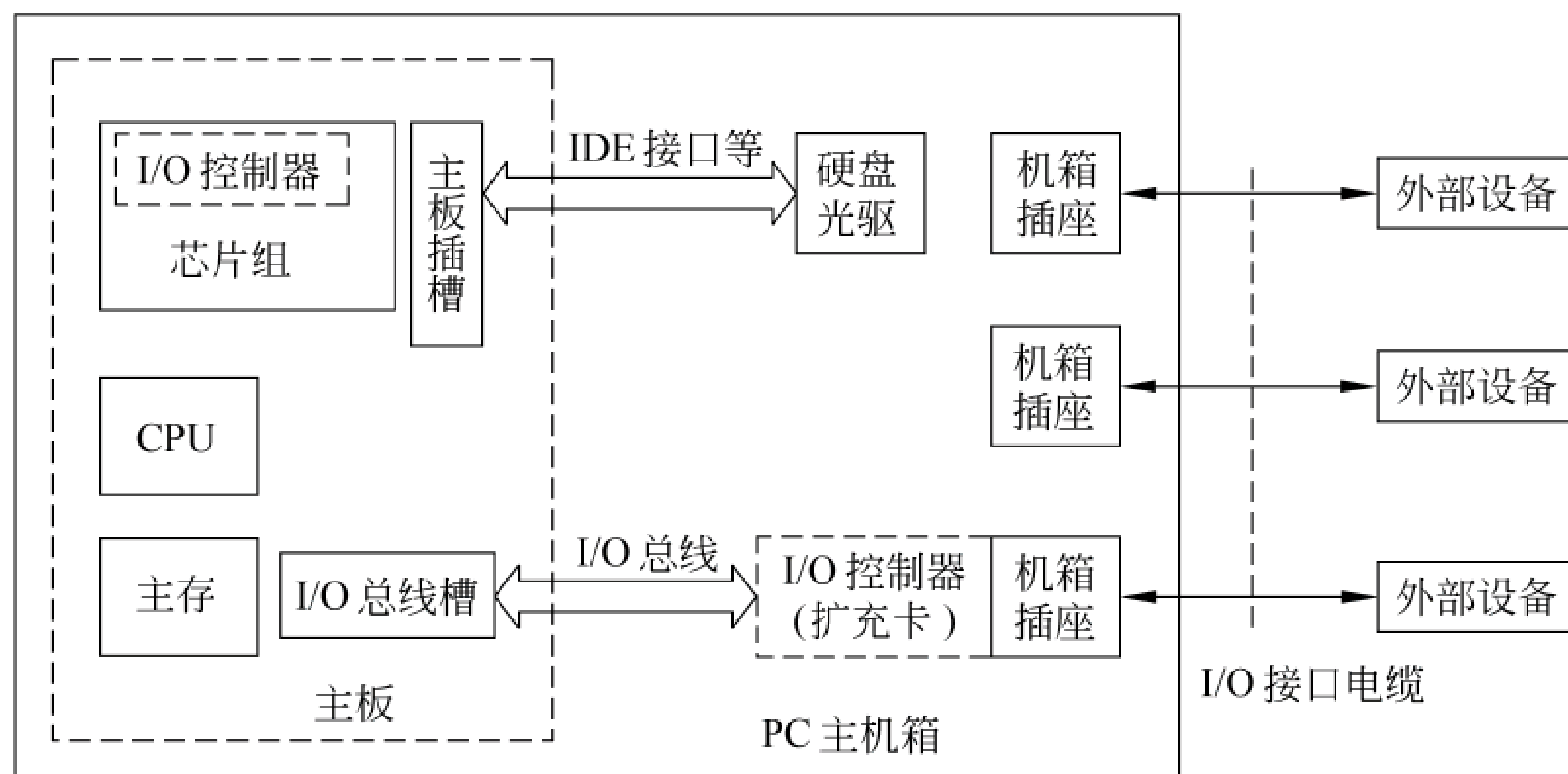


图 9.23　PC 中主机和外部设备的连接

源的手段，用户通过输入相应的命令或单击键盘和鼠标将 I/O 请求传递给操作系统；对于用户程序，操作系统提供了一组关于 I/O 的系统调用（如打开文件、读写文件、关闭文件等）。当用户程序需要从某个设备输入信息或将结果送到外设时，通过系统调用（以低级语言方式提供）或库函数调用（以高级语言方式提供），将 I/O 请求提交给操作系统。

操作系统通常把 I/O 软件组织成从高到低的 4 个层次，层次越低，则越接近设备而越远离用户程序。这 4 个层次依次为：

- 用户层 I/O 软件
- 与设备无关的操作系统 I/O 软件
- 设备驱动程序
- I/O 中断处理程序

大部分 I/O 软件都属于操作系统，但最初的 I/O 请求是在用户程序中提出来的。通常把提出 I/O 请求的用户程序看成是用户层 I/O 软件。例如，C 语言程序中的 printf 函数、scanf 函数等都是一种 I/O 请求。标准 I/O 库中包含了大量的涉及 I/O 的库例程。它们在用户程序中作为其中的一部分运行。

操作系统中与设备无关的 I/O 软件的基本功能是执行适用于所有设备的常用 I/O 功能，向用户层软件提供一个统一的调用接口。

设备驱动程序是与设备相关的 I/O 软件部分。每个设备驱动程序只处理一种设备或一类紧密相关的设备。每个设备都有一个相关的设备 I/O 接口（统称为 I/O 控制器），I/O 接口中有各种寄存器，包括控制寄存器、状态寄存器和数据缓冲寄存器等。设备驱动程序通过对这些寄存器进行编程，将相应的命令送控制寄存器，读取状态寄存器中的状态，从数据缓冲寄存器中读取或发送数据等。例如，对于磁盘，磁盘驱动程序知道磁盘控制器有多少寄存器、每个寄存器的用途以及进行磁盘操作所必需的全部参数，包括磁头定位时间、磁盘旋转时间、磁头数、磁道数、扇区数、交错因子等。一个用户的 I/O 请求，通过操作系统最终传递给设备驱动程序。所以，真正的 I/O 执行是由设备驱动程序完成的。例如，对于磁盘操作，磁盘驱动程序将完成：计算出请求块的物理地址、检查磁盘驱动器的电机是否运转、检测磁头是否定位在正确的柱面上等。

当设备驱动程序启动外设进行某种操作时，一种方式是采用等待外设完成相应的操作，

这种方式的效率很低，更有效的方式是主机向外设发出某个命令以后，转去执行其他程序，而当外设完成相应命令后用中断方式通知操作系统。此时调出相应的中断处理程序来对“外设完成任务”的事件进行处理。有关中断的概念将在本章的后面详细说明。

操作系统必须保证一个用户程序只能访问到该用户有权访问的外设部分。例如，如果文件的所有者没有给一个用户程序以访问的权限，那么操作系统则不允许该程序读或写磁盘上的这个文件。在一个具有共享外设的系统中，如果让用户程序直接执行 I/O 的话，则无法提供对外设的保护功能。这也是用户程序必须通过操作系统来使用外设的一个原因。

为了使操作系统能够直接和外设进行通信，并阻止用户程序直接访问外设，必须提供以下几种信息通信功能。

(1) 操作系统必须能向设备发出命令，这些命令不仅包含像读和写等主要操作，而且还包括对设备本身的一些操作，如磁盘寻道和旋转等待等。

(2) 当外设已经完成一个操作或遇到一个错误，则必须能够通知操作系统，以便作适当处理，例如当磁盘完成一次寻道后，它要能通知操作系统。

(3) 数据必须能在存储器和外设之间、CPU 寄存器和外设之间进行数据传输。

以上信息通信过程，操作系统最终是通过执行相应设备驱动程序中的一条条指令来完成的。因此在进行指令系统设计时，必须保证指令系统能提供对 I/O 进行访问的指令，这些指令被称为输入输出指令(I/O 指令)。

9.4.4 I/O 端口及其编址

系统如何在访问 I/O 的指令中标识要访问的 I/O 接口中的某个寄存器呢？这就是 I/O 端口的编址问题。I/O 端口实际上就是 I/O 接口中的寄存器，例如，数据缓存寄存器就是数据端口，控制/状态寄存器就是控制/状态端口。一个 I/O 端口可能是输入端口，输出端口，或者是双向的既可输入也可输出的端口。有些 I/O 端口用来存放数据，例如，串行接口中的发送和接收寄存器；还有些 I/O 端口用来控制外设，例如磁盘控制器中的控制寄存器。对这些端口的写或读操作即被认为是向 I/O 设备送出命令或从设备取得数据或状态。

为了便于 CPU 对 I/O 设备的快速选择和对 I/O 端口的方便寻址，必须给所有 I/O 接口中各个可访问的寄存器进行编址，有独立编址和统一编址两种方式。

1. 独立编址方式

独立编址方式对所有的 I/O 端口单独进行编号，使它们成为一个独立的 I/O 地址空间。这种情况下，指令系统中需要有专门的输入输出指令来访问 I/O 端口，输入输出指令中地址码部分给出 I/O 端口号，图 9.24 给出了独立编址方案示意图。

独立编址方式中的 I/O 地址空间和主存地址空间是两个独立的地址空间，因而无法从地址码的形式上区分，故需用专门的 I/O 指令来表明访问的是 I/O 地址空间。CPU 执行 I/O 指令时，会产生 I/O 读或 I/O 写总线事务，总线中也有专门的$\overline{IOR}$和$\overline{IOW}$控制信号线，通过 I/O 读写控制线，可以表明地址线上给定的是 I/O 端口号。通常 I/O 端口数比存储器单元少得多，选择 I/O 端口时，只需少量地址线，所以 I/O 端口译码简单，寻址速度快。使用专用 I/O 指令，使得程序清晰，便于理解和检查。但 I/O 指令往往只提供简单的传输操作，故程序设计灵活性差些。而且处理器必须提供两组读写命令($\overline{MEMR}$和$\overline{MEMW}$、$\overline{IOR}$和$\overline{IOW}$)，增加了总线控制逻辑的复杂性和处理器引脚数。此外，使用独立的地址空间还需要

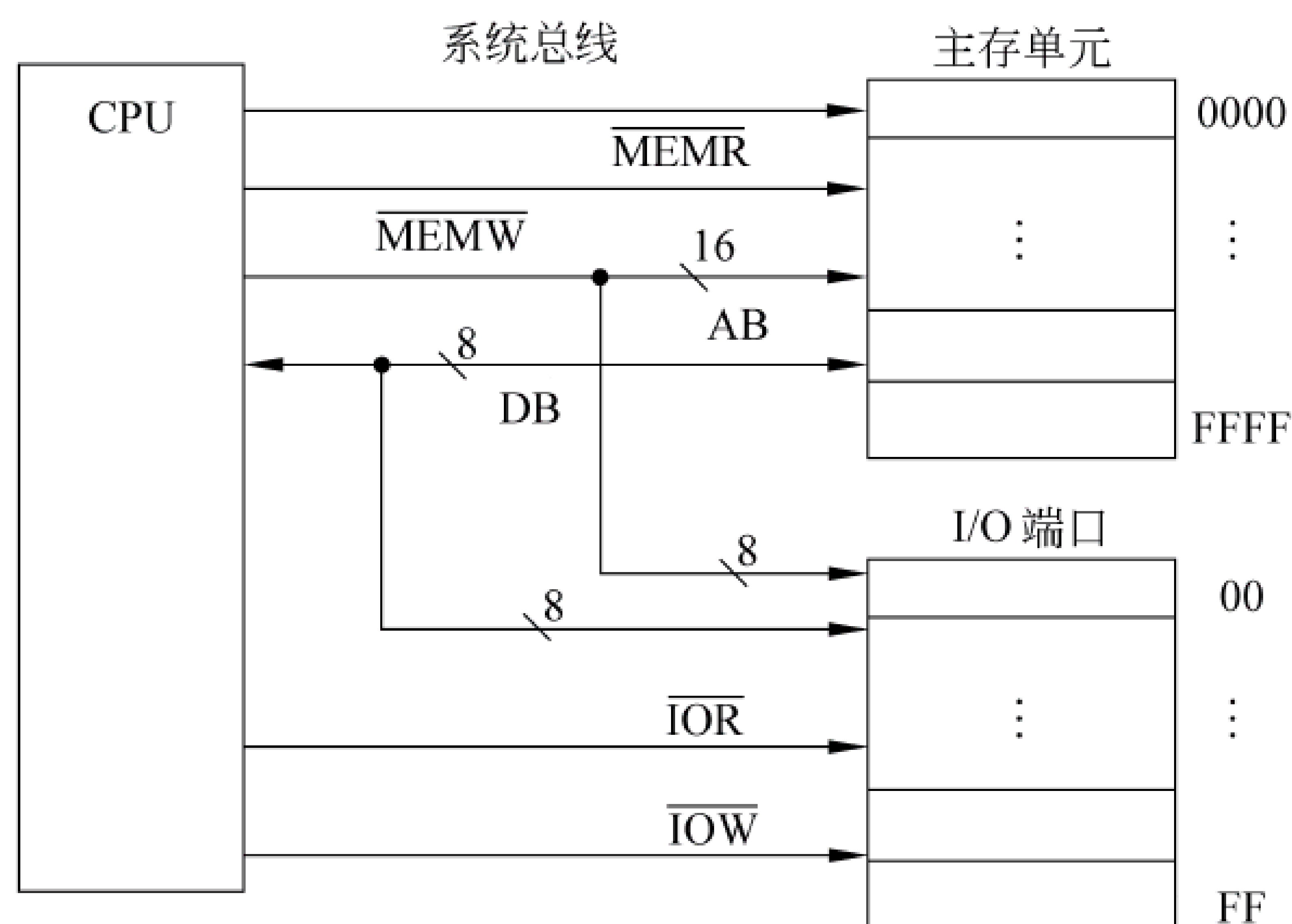


图 9.24　独立编址方案

专门的硬件保护机制。

例如，Intel 处理器就采用独立编址方式，其 I/O 地址空间由 2^{16}(64K)个地址编号组成，每个编号可以寻址一个 8 位的 I/O 端口。任何两个连续的 8 位端口可看成一个 16 位端口。该处理器提供专门的 I/O 指令：IN(INS)指令和 OUT(OUTS) 指令。执行这些指令时，处理器的 M/$\overline{\text{IO}}$引脚变成低电平，表示将出现对 I/O 地址空间进行访问的总线周期。在此访问周期中，处理器可以对 I/O 地址空间的某个设备进行 8 位、16 位或 32 位的数据传送。

2. 统一编址方式

统一编址方式下，I/O 地址空间与主存地址空间统一编址，即将主存地址空间分出一部分地址给 I/O 端口进行编号，因为 I/O 端口和主存单元在同一个地址空间的不同分段中，根据地址范围就可区分访问的是 I/O 端口还是主存单元，因而无须设置专门的 I/O 指令，只要用一般的访存指令就可以存取 I/O 端口。因为这种方法是将 I/O 端口映射到主存空间的某个地址段上，所以，也被称为“存储器映射方式”。Motorola 公司生产的处理器就采用该方案，图 9.25 给出了统一编址方案示意图。

统一编址方式下，当 CPU 执行到一个访存指令时，便生成一个存储器读或存储器写总线事务，因为不是专门的 I/O 指令，所以无法通过控制线来区分访问的是主存空间还是 I/O 空间。但是，通常划出来给 I/O 端口的地址部分是一段连续区域，因此这些地址有一定的特征，只要根据地址的某些特征就可区分访问的是主存单元还是 I/O 端口。例如，假定 I/O 空间在 8000H～FFFFH 范围内，说明 I/O 空间的特征是最高位 A15 为“1”。因而，可在设备 I/O 接口中，增加相应的控制逻辑，把 A15 和相应的$\overline{\text{MEMR}}$和$\overline{\text{MEMW}}$控制信号组合，形成对应的 I/O 读($\overline{\text{IOR}}$)和 I/O 写($\overline{\text{IOW}}$)信号，以控制对 I/O 端口的读写。

因为统一编址方式下 I/O 访问和主存访问共用同一组指令，所以它的保护机制可由分段或分页存储管理来实现，而无须专门的保护机制。这种存储器映射方式给编程提供了非常大的灵活性。任何对内存存取的指令都可用来访问位于主存地址空间中的 I/O 端口，并且所有有关主存的寻址方式都可用于 I/O 端口的寻址。例如，可用访存指令实现 CPU 寄存器和 I/O 端口的数据传送；可用 AND、OR 或 TEST 等指令直接操作 I/O 接口中的控制寄存器或状态寄存器。采用统一编址法的另一个好处是便于扩大系统吞吐率，因为外设或

I/O 寄存器数目除了受总存储容量的限制外几乎不受其他因素限制，这在大型控制或数据通信系统等特殊场合很有用。不过，因为 I/O 空间占用了一部分主存空间的地址，使得主存空间减少。此外，由于在识别 I/O 端口时全部地址线都需参与地址译码，使译码电路变复杂了，并需用较长时间译码，所以外设寻址时间变长了。

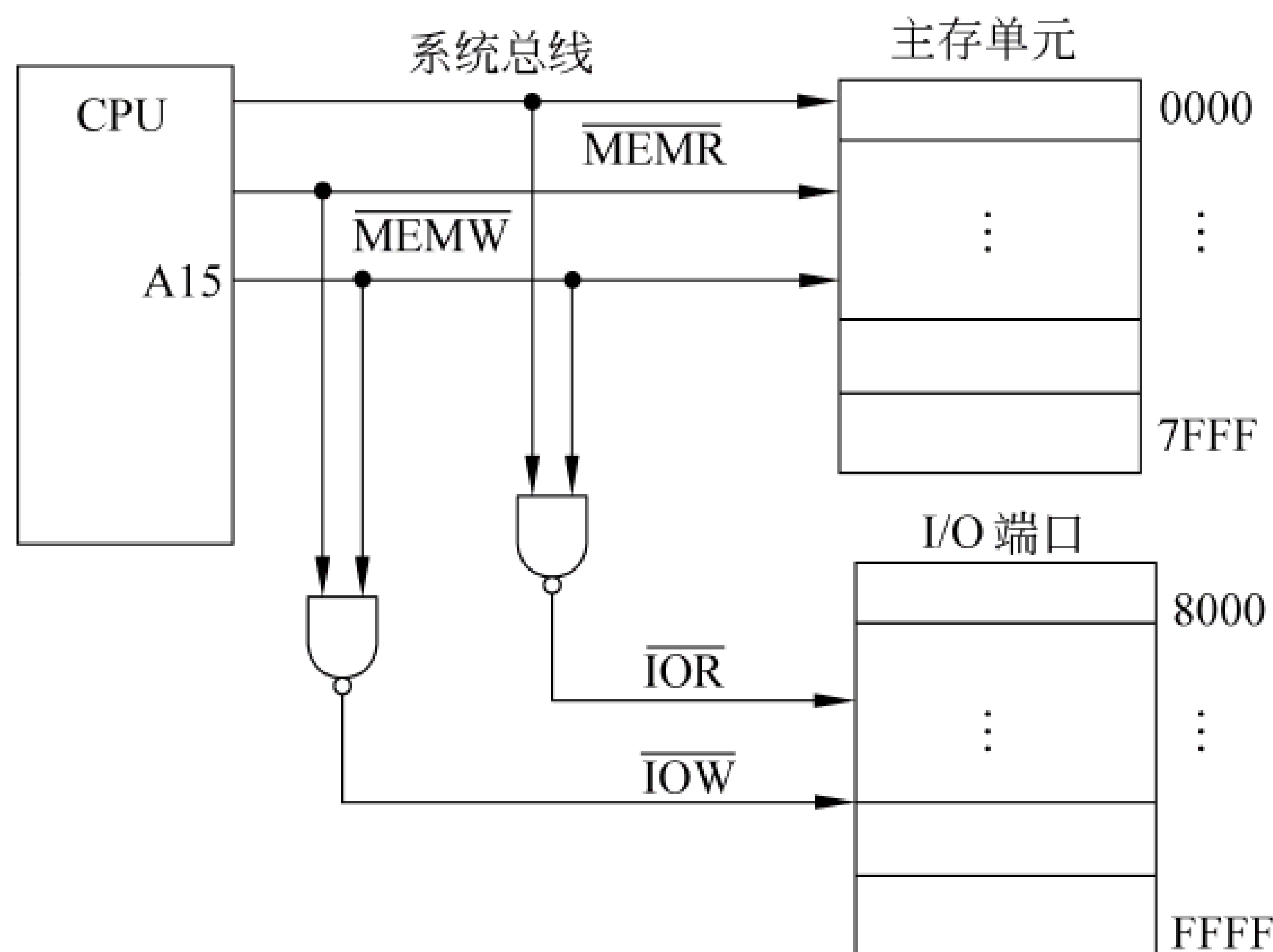


图 9.25 统一编址方式

按照数据传送的方式，输入输出设备可分为字符设备和块传送设备两种。一般块传送设备的数据直接和主存进行交换，CPU 中的寄存器通常只和字符设备进行数据交换。从要传送的数据来源或目的地来看，I/O 指令分为两类。

(1) 寄存器 I/O 指令。在寄存器和 I/O 端口之间传送一个单数据项(如字节、字或双字)。

(2) 成组 I/O 指令。在存储器和 I/O 端口之间传送一串数据项(如字节串、字串或双字串)。

例如，奔腾处理器提供的寄存器 I/O 指令有 IN 和 OUT，主要用于完成在 I/O 端口和 EAX 寄存器(32 位双字)、AX 寄存器(16 位字)或 AL 寄存器(8 位字节)之间的数据传送。奔腾处理器也提供成组 I/O 指令 INS 和 OUTS，它们完成一块内存区和 I/O 端口之间多个数据项的传送。INS 和 OUTS 指令中的端口号由 DX 寄存器给出，内存地址由(E)SI 或(E) DI 指定，每次传送后，(E)SI 或(E) DI 寄存器进行增量或减量运算，根据传送的数据项的大小增减 1(字节)或 2(字)或 4(双字)。

*9.4.5 I/O 接口的分类

I/O 接口有以下几种不同的分类方式。

(1) 按数据传送方式分，有并行接口和串行接口两类。对于主机侧的内部接口，数据在接口和主机之间总是通过系统总线按字节或字或多字进行并行传输；而在外设侧的外部接口，数据在接口和外设之间有串行和并行两种传输方式。并行接口在设备和接口之间同时传送一个字节或字的所有位(如 Intel 8255，SCSI，IDE 等)；串行接口则逐位地传送(如 Intel 8251，USB，IEEE 1394，SATA 等)。对于串行接口，接口内部必须有串-并转换部件。

(2) 按功能选择的灵活性来分，有可编程接口和不可编程接口两类。可编程接口能用

程序来选择或改变接口的功能和操作方式(如 Intel 8255,Intel 8251,Intel 8259A 等)。不可编程接口不能用程序来改变其功能,但可通过硬连线路逻辑来实现不同的功能(如 Intel 8212 等)。

(3) 按通用性来分,有通用接口和专用接口。通用接口可供多种外设使用,如 Intel 8255、Intel 8212;专用接口是为某类外设或某种用途专门设计的,如 Intel 8279 可编程键盘/显示器接口、Intel 8275 可编程 CRT 控制器接口等。

(4) 按数据传送的控制方式来分,有程控式接口、中断式接口和 DMA 式接口。程控式接口用于连接速度较慢的 I/O 设备,如显示终端、键盘、打印机等。现代计算机一般都可采用程序中断方式实现主机和外设之间的数据交换,所以大多数计算机中都配有中断式接口,如中断控制器 Intel 8259A。DMA 式接口用于连接如磁盘、磁带等高速设备,如 DMA 控制器 Intel 8257/8237A。有关中断式接口和 DMA 式接口的结构在后面介绍。

(5) 按设备的连接方式来分,有点对点接口和总线式多点接口。点对点接口只和一个外设相连,如打印机、键盘、调制解调器等设备。愈来愈重要的是多点方式,也称为总线式接口。这种多点接口的典型例子有 USB 接口、SCSI 接口和 IEEE 1394 接口。SCSI 接口是一种总线式并行接口,而 USB 和 IEEE 1394 接口则是总线式串行接口。

*9.4.6 并行传输和串行传输

对于 I/O 接口的主机侧,数据在接口和主机之间总是通过系统总线按字节或字或多字进行并行传输;而在 I/O 接口的外设侧,数据在接口和外设之间有并行和串行两种传输方式。

1. 并行传输方式

并行传输时数据在数据线上同时有多位一起传送,因此有多根数据线。通常,并行传输方式多用在同步总线中,因此,衡量并行总线速度的指标是最大数据传输率,即单位时间内在总线上传输的最大信息量。一般用每秒多少兆字节(MBps)来表示。例如,若并行总线的时钟频率为 33MHz,总线宽度为 32 位,每个时钟传输一个数据,则它的最大数据传输率为 $33\times32/8=132$MBps。数据传输率由总线时钟频率而得,在频率中的 1M 为 10^6,所以这里 $1M=10^6$,而不是 2^{20}。一般在数据传输率中出现的 M 和 G 等单位是以 10 为基来衡量的。

在并行传输方式中,由于所有并行传输的位信号必须有相同的定时信号来同步,当传输速度更快、传输线更长时,并行传输的实现变得越来越困难。因此,并行总线的时钟频率不可能提高很多,而串行总线只有一根数据线,进行传输的位之间不需要同步,因而可以有很高的传输速率,这就是为什么近年来许多 I/O 总线和 I/O 接口的传输方式都由并行转为串行的原因。

2. 串行传输方式

串行传输过程中有两个重要概念:波特率和比特率。在串行传输通道中,携带数据信息的信号单元叫码元,每秒钟通过信道传输的码元数称为波特率。即波特率所表示的是调制速度,是单位时间内传输线路上调制状态的变化数,单位为波特(band);比特率是指每秒钟通过信道传输的二进制位数,单位是位/秒(bps),比特率是数据传输率的一种度量方式。

波特率与比特率的关系为比特率=波特率×单个调制状态对应的二进制位数。

显然，两相调制(单个调制状态对应一个二进制位)的比特率等于波特率；四相调制(单个调制状态对应两个二进制位)的比特率为波特率的两倍；八相调制(单个调制状态对应三个二进制位)的比特率为波特率的三倍；依此类推。

串行传输只需一根数据线。串行传输时，按顺序传送一个数据的所有二进位，被传送的数据在发送部件中必须进行并行数据到串行数据的转换，这个过程称为拆卸；而在接收部件中则需要将串行数据转换成并行数据，这个过程称为装配。

在进行数据传送时，串行总线接口的发送端和接收端之间必须有时钟脉冲信号对传送的数据进行定位和同步控制。收/发时钟频率与波特率之间通常有简单的倍数关系。

$$收/发时钟频率=n\times 波特率$$

一般 $n=1,16,32,64$ 等。对于异步通信，常采用 $n=16$，对于同步通信，则必须取 $n=1$。图 9.26 给出了串行通信的简单原理图。

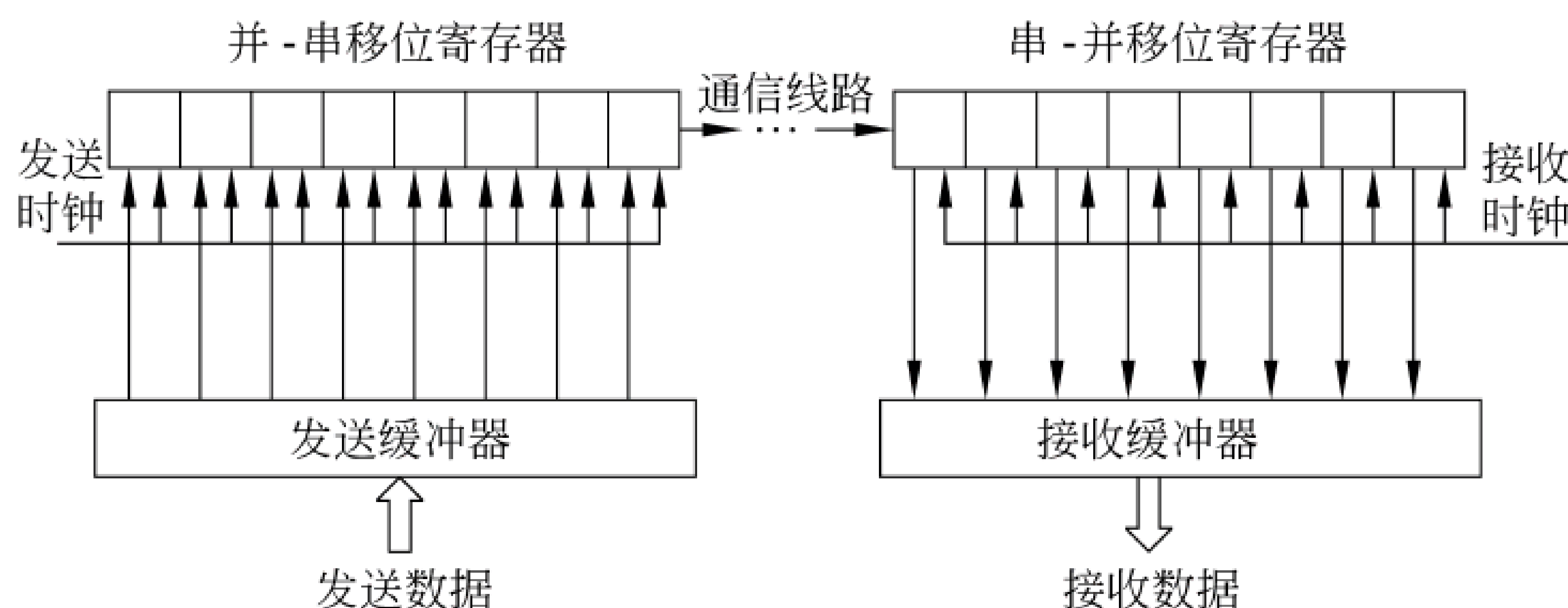


图 9.26　串行通信的简单原理

通信协议也叫通信规程，是指通信双方在信息传输格式上的一种约定。数据通信中，在收/发器之间传送的一位一位二进制的“0”或“1”，它们在不同的位上有不同的含义，有的可能是用于同步的信息位，有的是用于数据校验的位，也有的是一些控制信息或地址信息，所以传输的并不都是真正的数据信息。因此，在通信的双方必须在通信协议中事先约定好这些传输格式。串行方式下有异步和同步两种通信方式，它们有不同的通信协议。

(1) 异步串行通信协议

在异步串行通信中，每个字符作为一帧独立的信息，可以随机出现在数据流中，也就是说，每个字符出现在数据流中的时间是随机的、不确定的，接收端预先不知道。但每个字符一旦开始发送，收/发双方则以预先约定的固定时钟速率传送各位。因此，所谓异步主要体现在字符与字符之间的传送，同一字符内的位与位之间是同步的。为了使收/发双方在随机传送的字符与字符之间实现同步，通信协议规定在每个字符格式中设置起始位和停止位。协议规定每个字符格式由以下 4 个部分组成。

① 1 位起始位，总是低电平；

② 5～8 位数据位，紧跟在起始位后，规定从最低有效位开始传送；

③ 没有或一位奇偶校验位；

④ 1 位/1.5 位/2 位停止位，规定为高电平。

一般有效数据位为 5 位时，停止位取 1 位或 1.5 位，其他情况取 1 位或 2 位停止位。因此，一个字符可能由 7～12 位信息组成，称其为一个数据帧，数据传输格式如图 9.27 所示。

异步串行通信协议的格式中，起始位和停止位为异步字符传输的同步起着非常重要的

作用。由于同步只需在一个字符期间保持，下一个字符又可通过新的起始位和停止位进行同步，所以发送器和接收器不必使用同一个时钟，只需分别使用两个频率相同的局部时钟，使一个字符传送期间保持串行位的同步，就能保证整个线路上信息传输的正确。为了使接收器能够准确地发现每个字符的开始点，协议规定起始位和停止位必须采用相反的极性，利用前一个字符的停止位（高电平）到后一个字符的起始位（低电平）的负跳变，使接收器能很方便地发现一个字符的开始。为了保证一个字符到下一个字符的转换以负跳变开始，协议规定在字符与字符之间的空闲位也一律用和停止位一样的高电平来填充。空闲位的时间长度是任意的，不必是位时间的整数倍。

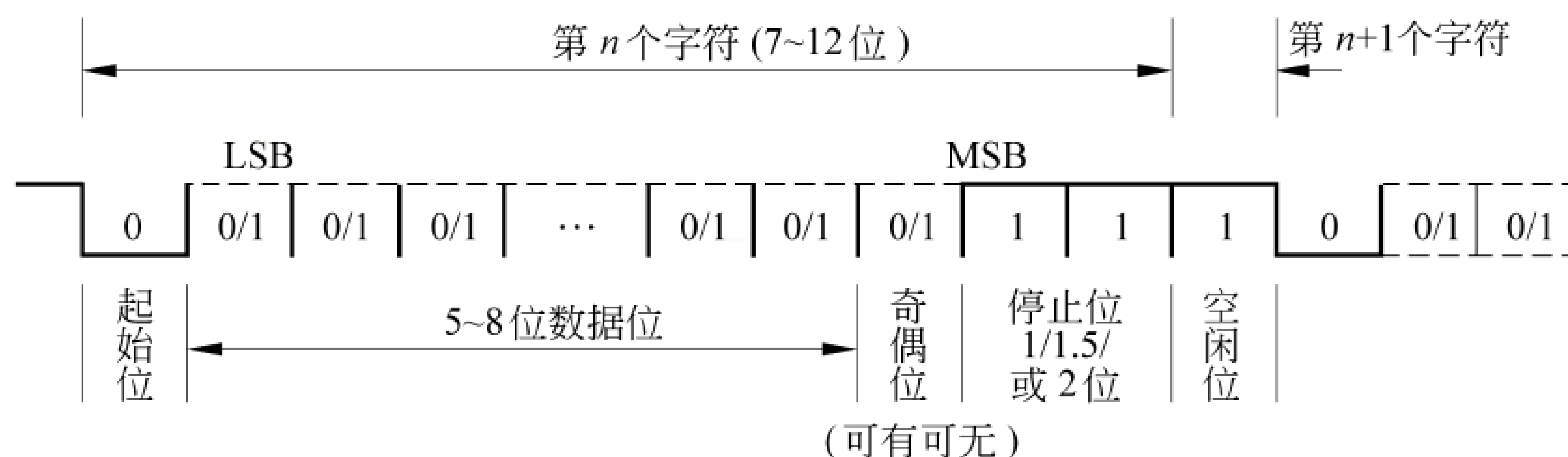


图 9.27　异步串行数据的格式

*（2）同步串行通信协议

在上述异步串行通信中，因为每个字符都有起始位和停止位，所以有效数据占整个信息的比例较小，假定数据帧的格式为 1 位起始位、7 位数据位、1 位校验位、1 位停止位，波特率为 1200bps，采用两相调制方式，则真正的数据传输率不是 1200 位/秒，而只有 840 位/秒。所以在一些数据速率要求较高的场合，需要采用另一种同步通信方式。

在同步串行通信中，数据流中的字符之间、每个字符内部的位与位之间都是同步的。这种通信方式对同步的要求非常严格，所以收/发双方必须以同一个时钟来控制数据的发送和接收。

同步传送的字符格式中没有起始位和停止位，不是用起始位来表示字符的开始，而是用同步字符来表示数据发送的开始。在发送端发送真正的数据字符之前，先发送同步字符去通知接收器，接收器在接收到同步字符后，便开始按双方约定的速率成批地连续接收数据，字符之间没有空隙。在发送器发送数据的过程中，如果出现数据没有准备好的情况，则发送器就发送同步字符来填充，直到下一个数据块准备好为止。

同步通信规程可分为面向字符型和面向比特型两大类。目前较通用的是面向比特型的通信规程，有 IBM 公司的同步数据链控制规程 SDLC 和国际标准化组织 ISO 的高级数据链控制规程 HDLC。这两个通信规程的基本原理和格式完全相同，仅在一些技术细节上有些区别。在 SDLC/HDLC 中，帧是信息传输的基本单元，所有信息均以帧的形式传输，既可以用于通信线路的控制，也可以用于数据传输。帧的结构如图 9.28 所示。

SDLC/HDLC 以一个起始标志和结束标志标识一个帧的开始和结束，这两个标志的编码都是 01111110，即两个 0 之间夹着 6 个 1。接收器用这个标志字符来建立帧的同步，我们把这个起始/结束标志称为同步字符。在起始标志后可以有一个地址场（A 场）和一个控制场（C 场）。用于传送地址信息和控制信息。在控制场后是信息场（I 场），用于传送数据信息。并不是每一帧都要有信息场，例如，用于线路控制的帧就不需要信息场。信息场的长度

可以是任意位长的信息位流，没有字或字符边界。也就是说信息场的长度可以不是字符的整数倍。在信息场之后是两个字节的CRC帧校验场(FC场)，在一帧中除了同步字符和自动插入的0以外，所有信息都参加CRC计算。采用SDLC/HDLC规程通信时，一帧内不应出现间隙，在两帧之间，发送器可发送连续的标志字符序列。

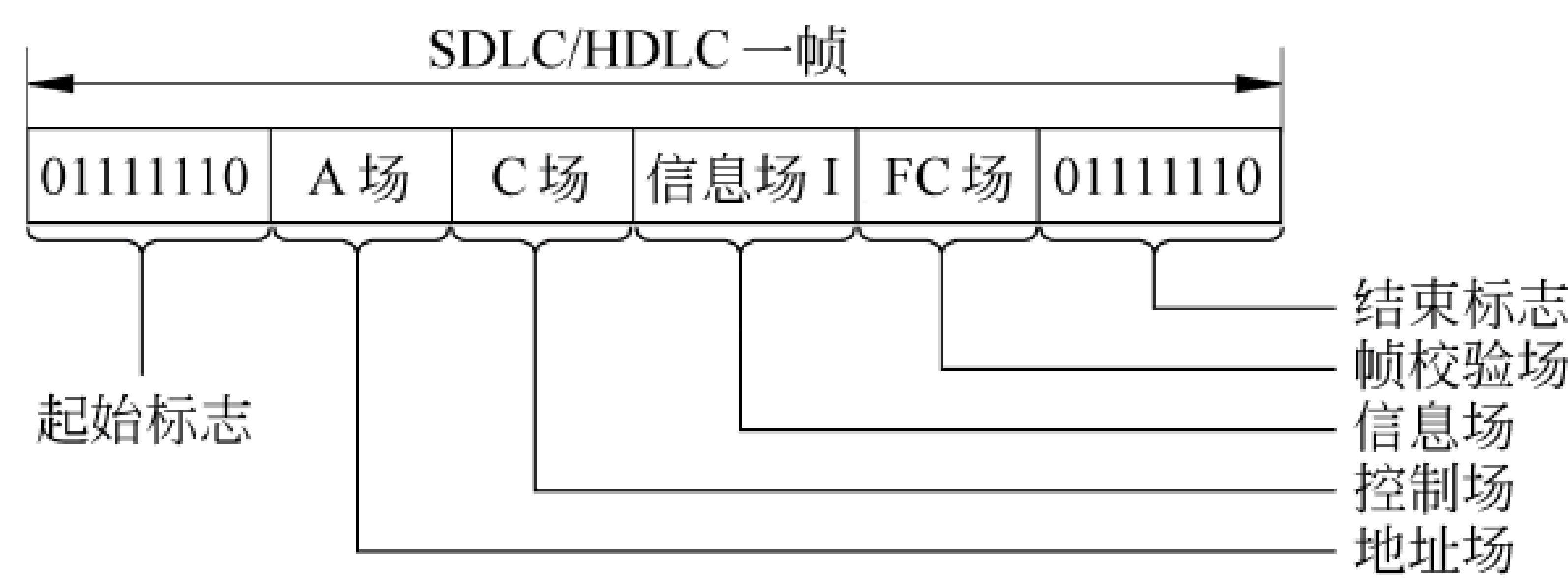


图9.28　SDLC/HDLC帧结构

*9.4.7　I/O接口举例

前面提到了各类I/O接口，为更好地理解I/O接口的功能和结构，以下选择了一个简单的通用并行接口和一个复杂的总线式接口作为代表来介绍其基本原理。下面具体介绍这两个I/O接口实例。

1. Intel 8255A并行接口芯片

Intel 8255A接口芯片是一种典型的可编程并行接口，可作为连接键盘、开关和喇叭等的接口，图9.29给出了其内部框图。

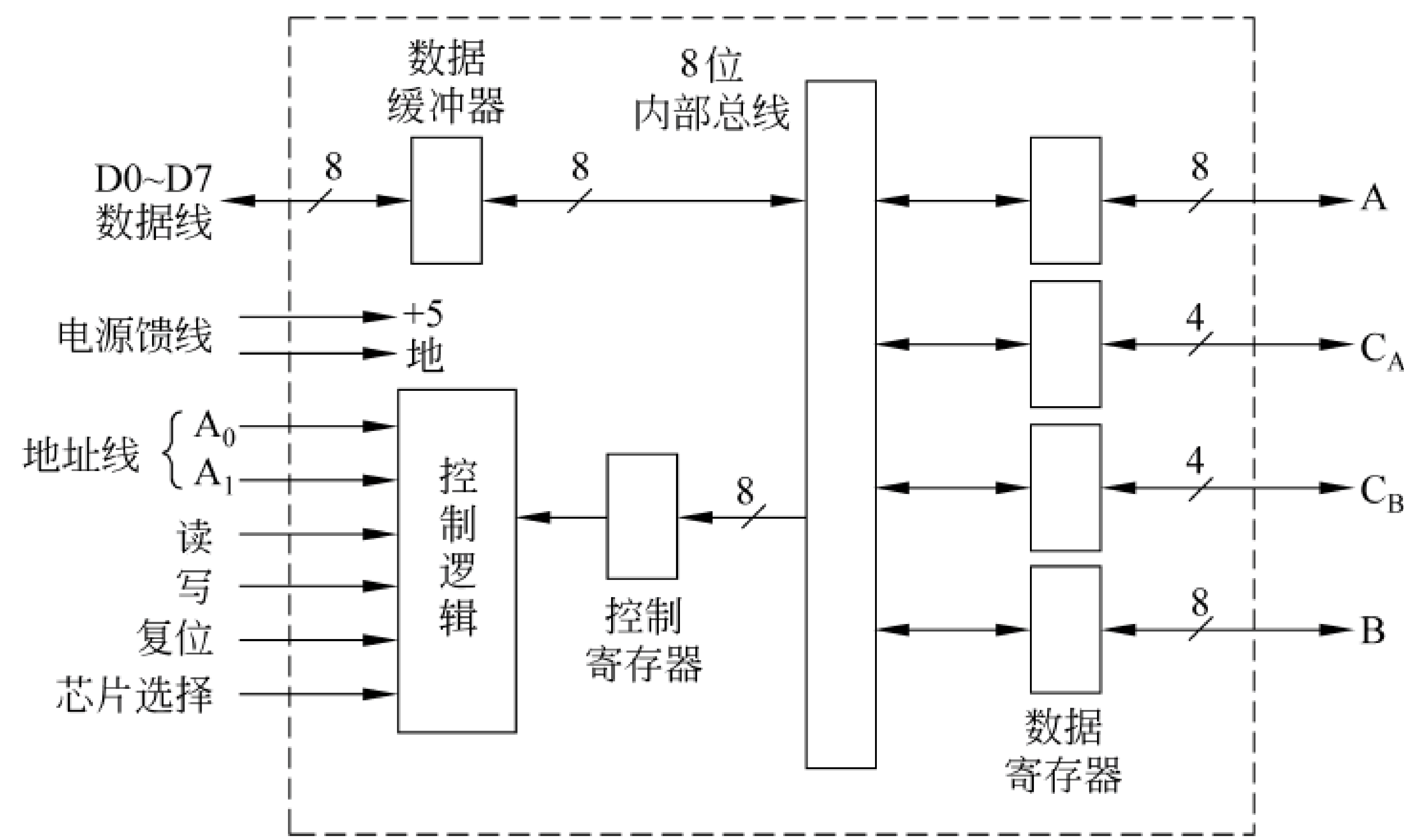


图9.29　Intel 8255A可编程并行接口

图9.29中右边是8255A的外设侧接口，24条外部数据线分成三组(A、B和C)，每组8根线，作为一个8位的I/O端口，可以实现8位数据的并行传送。也可以把A组和B组作为两个I/O端口，组C划分为两个4位组(C_A和C_B)，分别作为A组和B组的控制信号线。

图的左边是8255A的主机侧接口。它包括一组8位的数据线D0～D7，用于在主机和I/O端口之间传送数据(包括数据信息、状态信息或控制信息)。两根地址线用于指定A、B、C三个I/O数据端口和一个控制寄存器端口。

8255A 有三种工作方式(方式 0、方式 1 和方式 2),能使用多种数据传送方式(如无条件传送方式、程序查询方式和中断 I/O 方式,参见 9.5 节)完成 CPU 与 I/O 设备之间的数据交换。

方式 0 为基本输入输出方式,这种方式下不需要任何选通信号,A、B、C 三个端口都可以编程设定为输入或输出端口,用来实现无条件传送;方式 1 为选通输入输出方式,A、B 可分别编程为一个输入或输出端口,C_A 和 C_B 分别作为 A 和 B 的控制和定时(状态)信号线,可用来实现程序查询方式或中断 I/O 方式下的数据传送;方式 2 为双向输入输出方式,只有 A 口可编程为双向端口,C_A 作为 A 口的控制和定时(状态)信号线,用来实现程序查询方式或中断 I/O 方式下的数据传送。工作方式的设定是通过 CPU 执行输出指令(如 Intel x86 的 OUT 指令)向 8255A 内部的控制寄存器写一个工作方式命令字来实现的。

2. SCSI 总线式并行接口

SCSI(Small Computer System Interface,小型计算机系统接口)总线主要用于高性能的光驱、音频设备、扫描仪、打印机以及移动硬盘等的连接,是一种通用的总线式多对多连接接口。

如图 9.30 所示,挂接在 SCSI 总线上的设备以菊花链方式相连,设备之间是对等关系,而不是主从关系。每个 SCSI 设备有两个连接器,一个用于输入,一个用于输出。若干设备连接在一起,一端用一个终端器连接,另一端通过一块 SCSI 卡连到主机上,连在主机上的 SCSI 卡称为主适配器 HBA,可直接插到 PCI 插槽中,通过 PCI 总线与 CPU 相连,因此,SCSI 总线与 PCI 总线不在同一个层次。

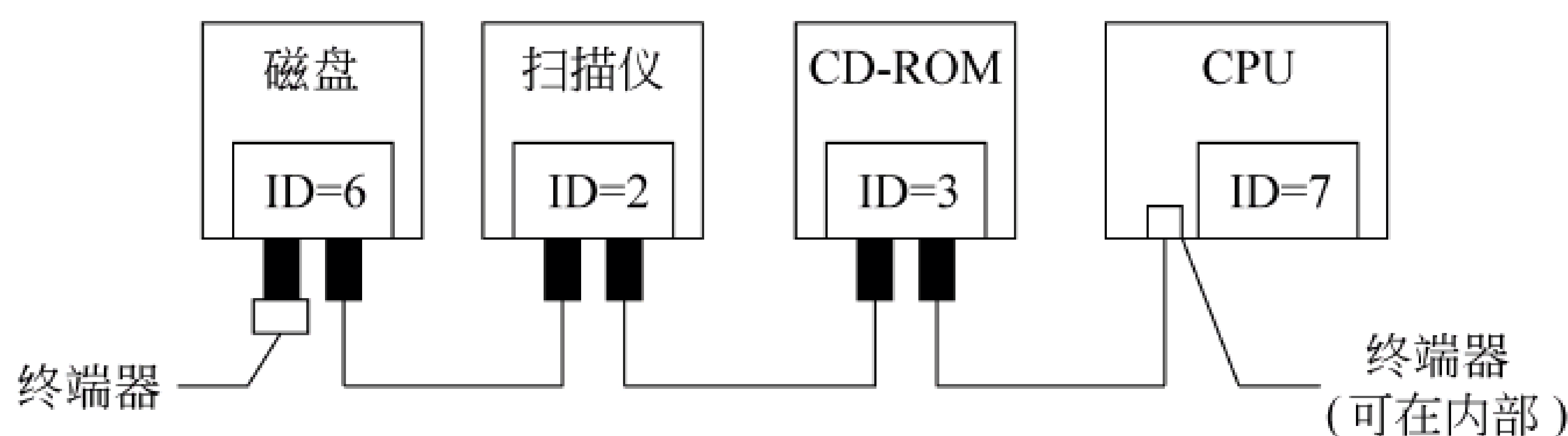


图 9.30 SCSI 设备的配置

SCSI 总线上的所有数据交换都是在请求方和目标方之间进行的。请求方和目标方依总线当时的运行状态来划分,而不是预先设定好的。但通常主机是请求方(即发起者),外设是目标方。总线上所有事务都按一定的顺序进行,依次经历以下各个总线阶段:

① 总线空闲阶段:表示没有设备使用总线,总线可用;

② 仲裁阶段:进行总线裁决,使一个设备获得总线使用权;

③ 选择阶段:让请求方选择一个目标设备来执行某个功能;

④ 重新选择阶段:允许目标设备重新连接请求方,以恢复原先由请求方启动而被目标设备挂起的操作。例如,磁盘进行寻道时,无须占据总线,此时,磁盘等目标设备可以先释放总线,让出总线给其他设备,等到准备好读写数据时,再进入重新选择阶段,所以,SCSI 总线支持事务分离方式。一旦在发起者和目标设备之间建立了连接,则可进行信息传送。可以由发起者向目标发出命令,或由目标向发起者回送状态,也可以在发起者和目标之间传送数据或消息。

⑤ 命令阶段:传送命令信息,使目标设备从请求方得到命令;

⑥ 数据阶段：目标设备请求数据传送。在该阶段可以进行数据输入(目标方到请求方)或数据输出(请求方到目标方)操作；

⑦ 状态阶段：目标设备向请求方发送状态信息；

⑧ 消息阶段：目标设备请求传送一个或多个消息。在该阶段可以进行消息输入(目标方到请求方)或消息输出(请求方到目标方)。

SCSI 总线中各总线事务在正常情况下按上述各阶段的顺序进行，当出现某种条件时，发起者和目标之间通过消息互换，引起总线阶段的转换。SCSI 总线阶段状态转换如图 9.31 所示。

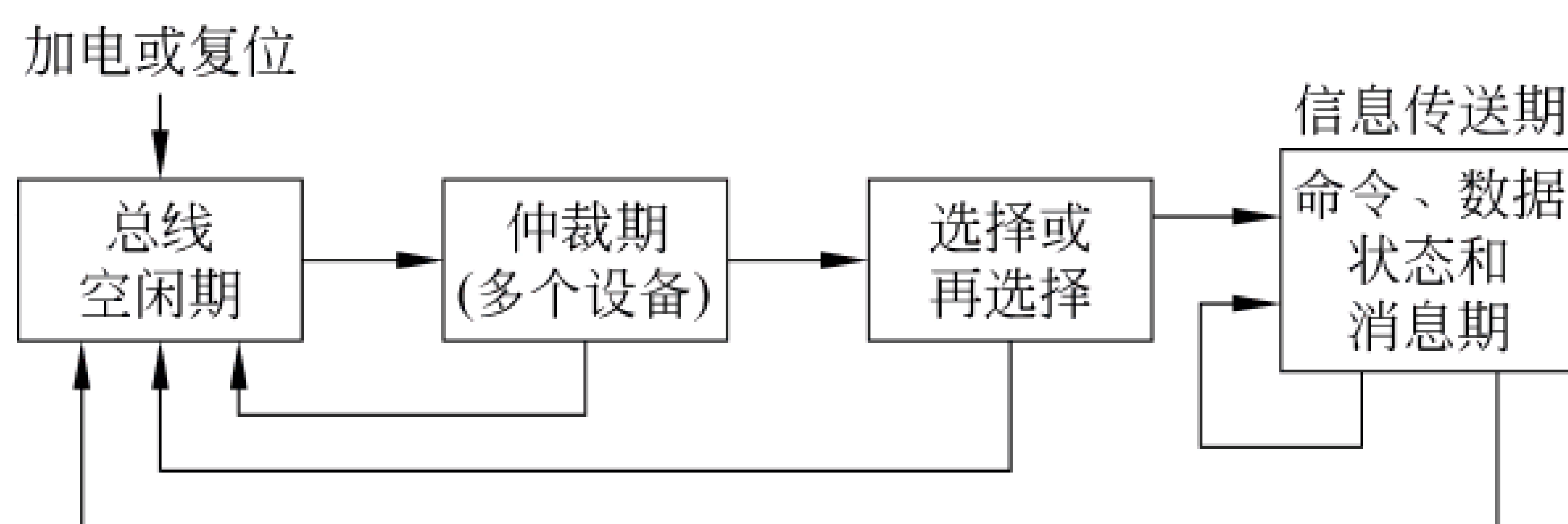


图 9.31　SCSI 总线阶段状态转换图

最早的 SCSI 规范 SCSI-1 只提供 8 位数据线，采用异步通信或 5MHz 的同步通信，最多允许 7 个设备以菊花链方式连接到主机上。1991 年出版了修改后的规范 SCSI-2，数据线可选择扩展到 16 位或 32 位，采用同步通信，时钟速度增加到 10MHz，所以最大数据传输率为 20MBps 或 40MBps。SCSI-3 允许连接 16 个设备，其数据传输率更高。后来发展的串行 SCSI 的数据传输率达 640Mbps(电缆)或 1Gbps(光纤)。

SCSI-1 规范规定了总线的信号系统共有 50 条信号线，采用 50 针扁平电缆或双绞线，称为 SCSI A 电缆。其中有 9 条数据线(8 条数据和一条奇偶校验)和 9 条控制线，其余为地线或电源线，9 根控制线如下。

BSY：由使用总线的设备来设置，表示自己使总线处于忙状态。

SEL：发起者选择目标时设置，或目标重新选择发起者时设置。

C/D：目标用来标识数据线上是控制信息(命令、状态或消息)还是数据信息。

I/O：目标用来标识数据传送的方向是输入(Input)还是输出(Output)。

MSG：目标用来标识正在向发起者传送的是消息。

REQ：目标准备好后用来请求数据传送。此时，发起者将接受来自总线的数据(数据输入阶段)，或把数据传送到总线(数据输出阶段)。

ACK：发起者用来应答目标的 REQ 请求。表示正在进行相应的数据传送操作。

ATN：发起者用来通知目标，说明它将有消息可传送。

RST：使总线复位。

SCSI-2 增加了 24 位数据线和相应的三个奇偶校验信号线以及其他控制线和地线及电源线，因而 SCSI-2 在 SCSI A 电缆的基础上增加了 68 针 B 电缆。

SCSI 总线中，信息传送的类型有命令输出、数据输入、数据输出、状态输入、消息输入和消息输出。这些命令、数据、状态和消息都是通过数据线传输的。总线通过相应的控制线来区别数据线上传输的信息类型，其定义如表 9.1 所示。

表 9.1 各信息传送阶段的定义

阶 段	C/D	I/O	MSG	DB7-0,P	传 送 方 向
数据输出	0	0	0	数据	发起者→目标
数据输入	0	1	0	数据	目标→发起者
命令	1	0	0	命令	发起者→目标
状态	1	1	0	状态	目标→发起者
消息输出	1	0	1	消息	发起者→目标
消息输入	1	1	1	消息	目标→发起者

SCSI 总线的裁决采用自举分布式方案(参见第 8 章 8.2.4 节)。总线中的每个设备(一个主机和另外最多 7 个设备)都有一个唯一的标识号 ID(0～7),这个标识号 ID 就是设备对应的优先级,7 为最高,0 为最低。需要使用总线的设备在仲裁阶段启动一根与该设备 ID 对应的数据线使之有效,每个设备通过查看相关的数据线来确定是否将获得总线的使用权。图 9.32 给出了一个常用的 SCSI 总线时序。它描述了从目标设备读取数据并送发起者的总线事务过程。

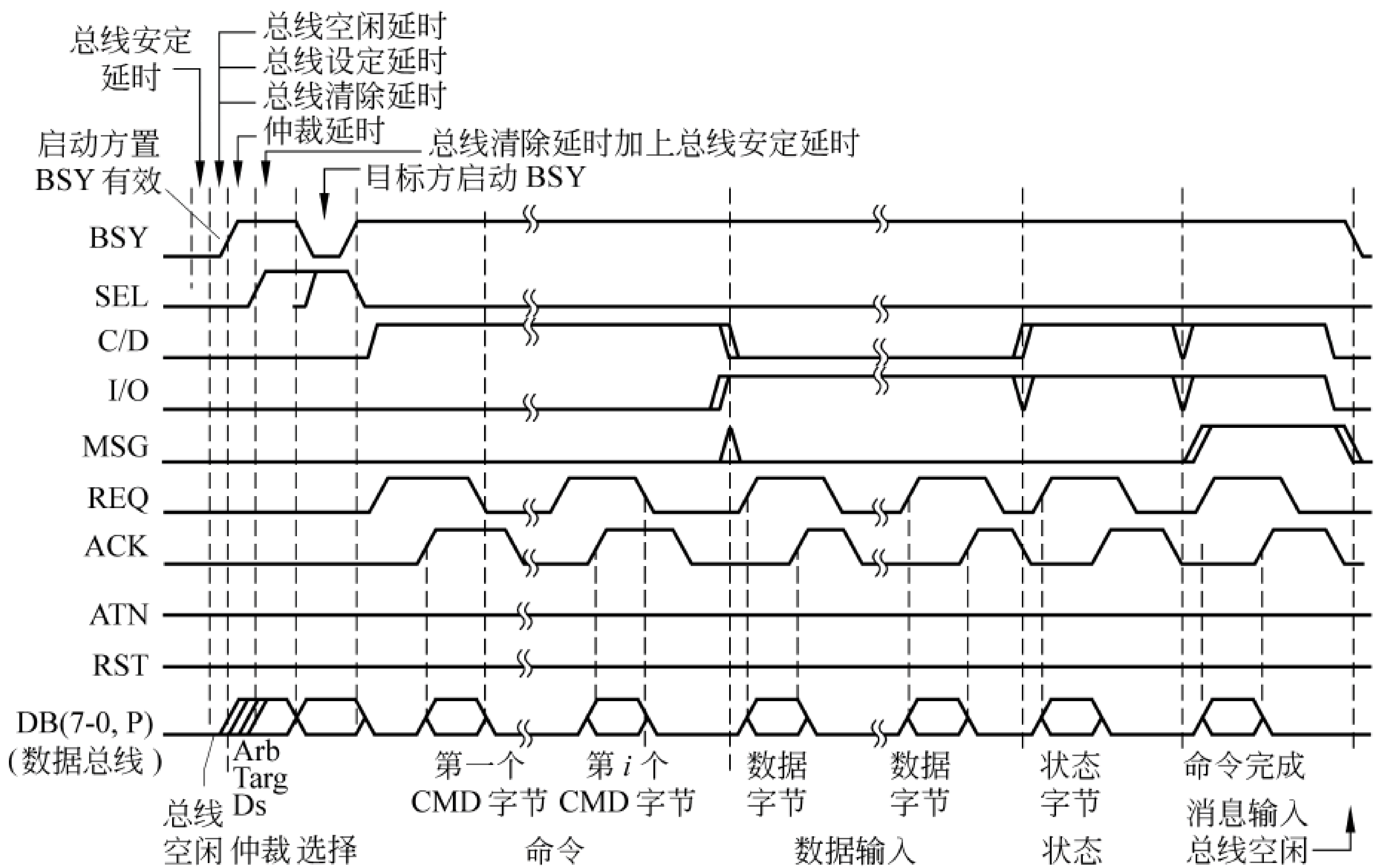


图 9.32 SCSI 时序举例

从目标设备读取数据送到发起者的整个过程如下:

① 开始时,总线处于空闲状态。

② 仲裁阶段:要求使用总线的设备在相应的数据线上置请求信号。各设备通过查看有无优先级比自己高的设备的请求来确定自己是否能占用总线。一旦某设备获胜,则将成为发起者,通过启动 SEL 信号进入选择阶段。

③ 选择阶段:发起者使与自身 ID 和目标 ID 对应的两根数据线有效,并在一定延迟后,使 BSY 信号无效。目标设备识别 ID 后,当检测 SEL 信号有效,而 BSY 和 I/O 无效时,

它使 BSY 信号有效。当发起者检测到 BSY 信号有效时,释放数据线,并取消 SEL 信号。

④ 命令阶段:目标设备通过启动 C/D 线有效,表示已进入命令阶段。此时,它将 REQ 信号设为有效,表明它请求发起者传送命令的第一个字节。发起者在送出第一个命令字节后,使 ACK 有效。目标设备读入一个命令字节后,取消 REQ 信号,然后发起者也取消 ACK 信号。命令的其他字节用同样的 REQ/ACK 握手信号来传送。

⑤ 数据输入(出)阶段:目标接收和解释命令后,取消 C/D 信号,使进入数据输入(出)阶段,数据传送的方向由信号 I/O 标识。通过 REQ/ACK 握手信号进行数据传送。

⑥ 状态阶段:目标使 C/D 信号有效,结束数据阶段而进入状态阶段(此时 I/O 信号有效),通过 REQ/ACK 握手信号进行状态信息的输入。

⑦ 消息阶段:目标设备使 MSG 线有效,以进入消息阶段。正常情况下,会传送一个"命令完成"消息给发起者。发起者收到该消息后,就释放总线,使其空闲。

9.5 I/O 数据传送控制方式

I/O 数据传送控制方式分为以下 4 种。

1. 程序直接控制 I/O 方式

直接通过查询程序来控制主机和外设之间的数据交换。因此也称为查询或轮询(polling)方式。该方式在查询程序中安排相应的 I/O 指令,通过这些指令直接向 I/O 接口传送控制命令,并从 I/O 接口中取得外设和接口的状态,根据状态来控制外设和主机的数据交换。

2. 程序中断 I/O 方式

程序中断方式的基本思想是,当 CPU 需要进行输入输出时,先执行相应的 I/O 指令,将启动命令发送给相应的 I/O 接口和外设,然后 CPU 继续执行其他程序。I/O 接口接收到 CPU 送过来的命令后,就开始启动外设进行相应的操作,当外设和 I/O 接口完成了 CPU 交给的任务后,I/O 接口便向 CPU 发中断请求。CPU 响应后,就中止正在执行的程序,转入一个"中断服务程序"。在"中断服务程序"中完成数据传送任务,传送完毕后再回到被中断的原程序继续执行。这种方式下,通常每次中断只能交换一个数据。

3. 直接存储器存取 I/O 方式

直接存储器存取(Direct Memory Access)方式,简称 DMA 方式,主要用于高速设备(如磁盘、磁带等)和主机间的数据传送,这类高速设备采用成批数据交换方式,且单位数据之间的时间间隔较短。DMA 方式的基本思想是,在外设和主存之间直接进行数据传送,用一个专门的硬件(DMA 控制器)来控制总线进行数据交换。在进行 DMA 传送时,CPU 让出总线控制权,由 DMA 控制器控制总线。DMA 控制器通过"窃取"一个主存周期完成和主存之间的一次数据交换,或独占若干个主存周期完成一批数据的交换。

4. 通道和 I/O 处理器方式

一般微型机采用前面三种方式。对于大型计算机系统来说,为了获得 CPU 和外设之间更高的并行性,也为了让种类繁多、物理特性各异的外设能以标准的接口连接到系统中,通常采用自成独立体系的通道结构或 I/O 处理器。在它进行主存和外设之间的信息传送时,CPU 执行自己的程序,两者完全并行。

9.5.1 程序直接控制I/O方式

程序直接控制方式直接通过查询程序来控制主机和外设之间的数据交换，通常有以下两种类型。

1. 无条件传送方式

也称同步传送方式，主要用于对一些简单外设（如开关、继电器、7段显示器或机械式传感器等）在规定的时间用相应的I/O指令对接口中的寄存器进行信息的输入或输出。其实质是通过程序来定时，以同步传送数据，适合于各类巡回检测采样或过程控制。图9.33是一个采用无条件传送方式的接口示意图。图中接口中有一个数据锁存器和一个三态缓冲器，它们共用同一个地址，可以看成是同一个输入输出数据寄存器。通过相应的I/O指令可直接对该寄存器进行访问，在读写信号的控制下进行数据的输入和输出。由此可见，无条件传送的接口比较简单，无须任何定时信号和状态查询，只需要进行相应的读写控制和地址译码即可，9.4节介绍的Intel 8255A中的方式0即是典型的无条件传送接口。

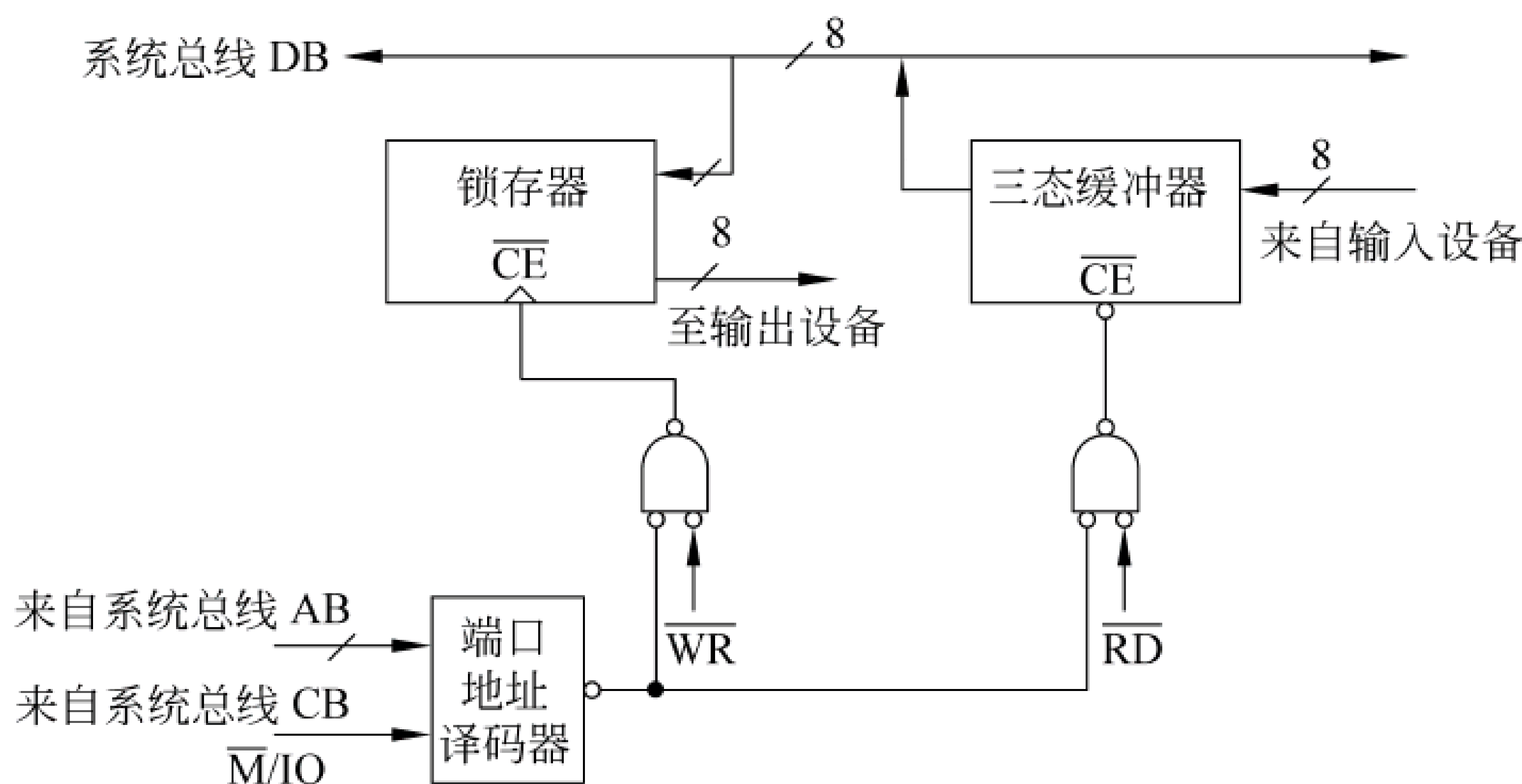


图9.33 无条件传送接口

无条件传送方式下，处理器对外设接口进行周期性的定时访问，直接对I/O端口进行数据存取。因此，这种方式下，处理器在I/O操作上的时间开销多少与定时访问的时间间隔有关。对于慢速设备，因为定时访问时间间隔长，所以，I/O操作所用的处理器时间占整个处理器时间的比例较少，对处理器效率影响不大；而对于快速设备，因为需要频繁地进行I/O访问，所以，很多处理器时间被I/O操作占用，因而这种方式不宜用于高速设备的I/O。

2. 条件传送方式

条件传送方式也称为异步传送方式。对于一些较复杂的I/O接口，往往有多个控制、状态和数据寄存器，对设备的控制必须在一定的状态条件下才能进行。此时，可通过在查询程序中安排相应的I/O指令，由这些指令直接从I/O接口中取得外设和接口的状态，如“就绪（Ready）”、“忙（Busy）”、“完成（Done）”等，根据这些状态来控制外设和主机的信息交换。因此这是一种通过查询接口中的状态来控制数据传送的方式，所以也被称为程序查询方式，其接口结构如图9.34所示。

图9.34所示的I/O接口中，左边是系统总线侧。CPU通过执行I/O指令来访问I/O接口。通过系统总线向I/O接口送出“启动”命令，读取“就绪”等状态信息，并向（从）数据

缓冲寄存器写入(读取)数据。I/O接口的右边是和设备相连的电缆或接口插座侧,可以送出“启动设备”命令,或接受“设备工作结束”信号,并可通过数据线与设备交换数据信息。CPU采用条件传送方式通过该I/O接口读取外设数据的过程如下:①CPU执行相应的I/O指令向该接口送出“启动”命令,设备选择电路对CPU送出的地址进行译码,选中本I/O接口,这样,与非门输出信号使“完成”状态触发器D清0,而使“启动”命令触发器B置1;②I/O接口通过连接电缆向外设发送“启动设备”命令;③外设准备好一个数据,通过电缆向I/O接口中的数据缓冲寄存器输入数据;④外设向I/O接口回送“设备工作结束”状态信号,使状态触发器D置1,使命令触发器B清0;⑤CPU通过执行指令不断读取I/O接口状态,因触发器D已被置1,故CPU查询到外设“准备就绪”;⑥CPU通过执行I/O指令从数据缓冲寄存器读取数据。通过以上6个步骤,CPU和外设之间完成一次数据交换过程。

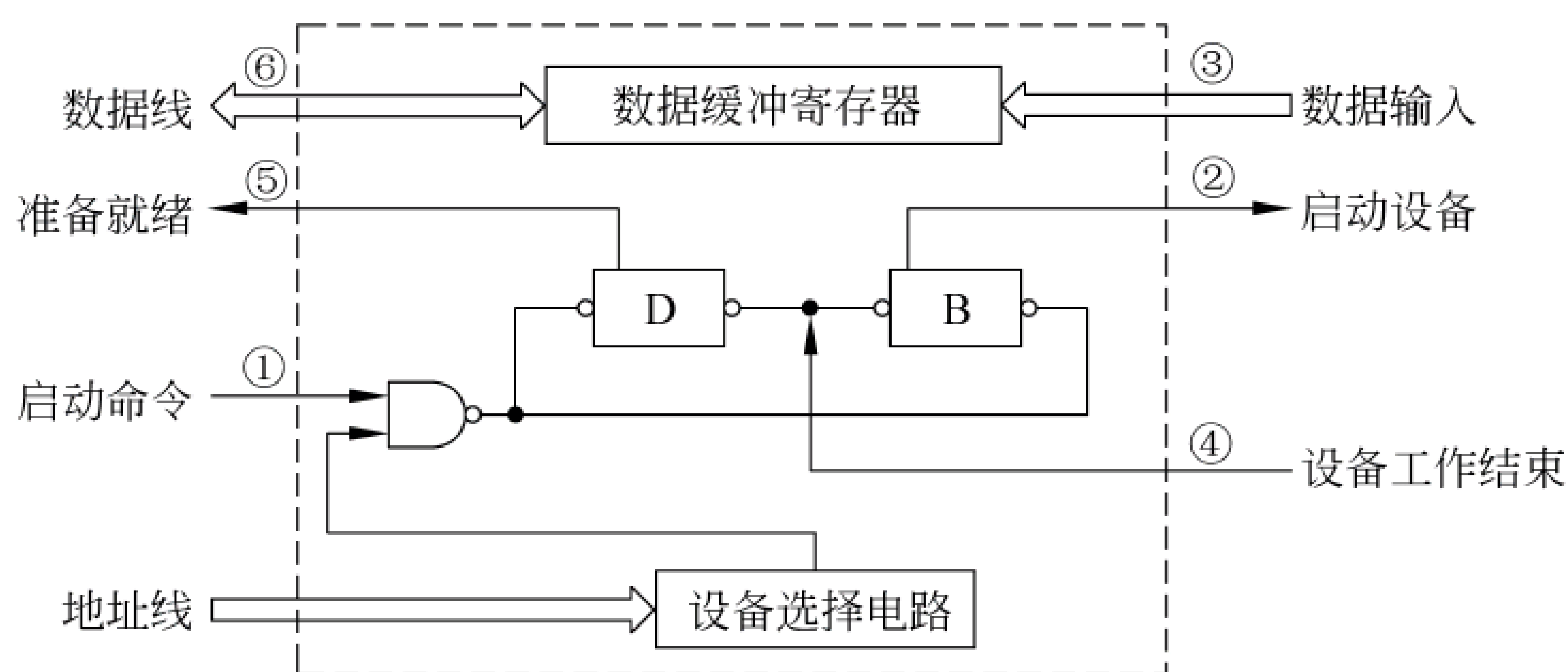


图9.34　条件传送方式接口

设备是否适合采用条件传送方式,主要取决于I/O设备本身的特点以及设备是否能够独立启动I/O等。键盘和鼠标等是随机启动的低速I/O设备,当用户按下键盘、移动鼠标或单击鼠标按钮时,便启动了I/O设备的输入操作。对于这类自身独立启动I/O的设备,虽然可以采用定时程序查询方式,但是,由于设备的启动是由用户随机进行的,所以,有可能用户长时间没有输入而引起查询程序长时间等待,从而降低处理器的使用效率;因此,这类设备大多采用中断方式进行I/O。对于由操作系统启动的设备,则只有被操作系统激活后才需要查询。像磁盘、磁带和光盘存储器等成块传送设备一旦被启动,便可连续不断地传送一批数据,处理器无须对每个数据的传送进行启动,而且每个数据之间的传输时间很短,如果用定时查询方式,则会因为频繁查询而使处理器为I/O操作所花费的时间比例变得很大,因此,不适合采用程序查询方式。像针式打印机等字符类设备,每个字符之间的传输时间较长,并且每传送一个字符需要启动一次,因而可以采用程序查询方式。

根据被启动查询的方式的不同,条件式程序查询方式有两种:定时查询和独占查询。可根据外设的特点选择采用定时查询方式和独占查询方式。

定时查询指周期性地查询接口状态,一旦进入查询,则总是一直等到条件满足,才进行一个数据的传送,传送完成后返回到用户程序。定时查询的时间间隔与设备的数据传输率有关。下面举例说明对于不同I/O传输速率的设备,其CPU为I/O操作所花费的时间开销是不同的。

例9.1　假定查询程序中所有操作(包括读取并分析状态、传送数据以及重启用户程序等所有步骤)所用的时钟周期数至少是400个,处理器的主频为500MHz,即处理器每秒钟

产生 500×10^6 个时钟周期。假定设备一直持续工作，采用定时查询方式，则以下三种情况下，CPU用于I/O的时间占整个CPU时间的百分比各是多少？

(1) 鼠标必须每秒钟至少被查询30次，才能保证不错过用户的任何一次移动。

(2) 软盘按16位为单位传送数据，数据传输率为50kBps，要求没有任何数据丢失。

(3) 硬盘以16字节为单位传送数据，数据传输率为4MBps，要求没有任何数据丢失。

解：对于查询方式，CPU用在输入输出上的时间由查询次数乘上查询操作时间得到。

(1) 对于鼠标，每秒钟内用于查询的时钟周期数至少为 $30\times400=12000$，因此，鼠标查询所花费的处理器时钟周期的百分比至少为 $12000/(500\times10^6)\approx0.002\%$。显然，CPU对鼠标的查询操作对性能的影响不是很大。

(2) 对于软盘，因为每次数据传送的单位为16位，占2个字节，所以只有当查询的速率达到每秒50kB/2B=25k次时才能保证没有任何数据被丢失。因此，每秒钟内用于查询的时钟周期数为 $25\text{k}\times400$，在整个CPU时间中所占的百分比为 $25\text{k}\times400/(500\times10^6)=2\%$。这个开销非常大，但在只有少量像软盘这样的I/O设备的低端系统中，这个开销是可以忍受的。

(3) 对于硬盘，要求每次以16字节为单位进行查询，所以查询的速率应达到每秒4MB/16B=250k次的速度，故每秒钟内用于查询的时钟周期数为 $250\text{k}\times400$，用于硬盘输入输出的时间占整个CPU时间的百分比为 $250\text{k}\times400/(500\times10^6)=20\%$ 。也就是说，CPU的五分之一时间被用于查询硬盘，显然，对硬盘用查询方式是不可取的。

如果软盘和硬盘仅有25%的时间是活动的，那么，操作系统只要在设备被激活时进行查询，所以查询的平均开销将被分别降到0.5%和5%。

有时因为操作系统不知道什么时候设备会响应并准备好一次传送，所以，一旦操作系统发出一个对设备的启动命令，它就必须接连不断地查询。这种一旦设备被启动，CPU就一直持续对设备进行查询的方式，称为独占查询方式。

独占查询方式下，CPU被独占用于某设备的I/O，它完全控制I/O整个过程，也即CPU花费100%的时间在I/O操作上，此时外设和CPU完全串行工作，其查询程序的流程如图9.35所示。

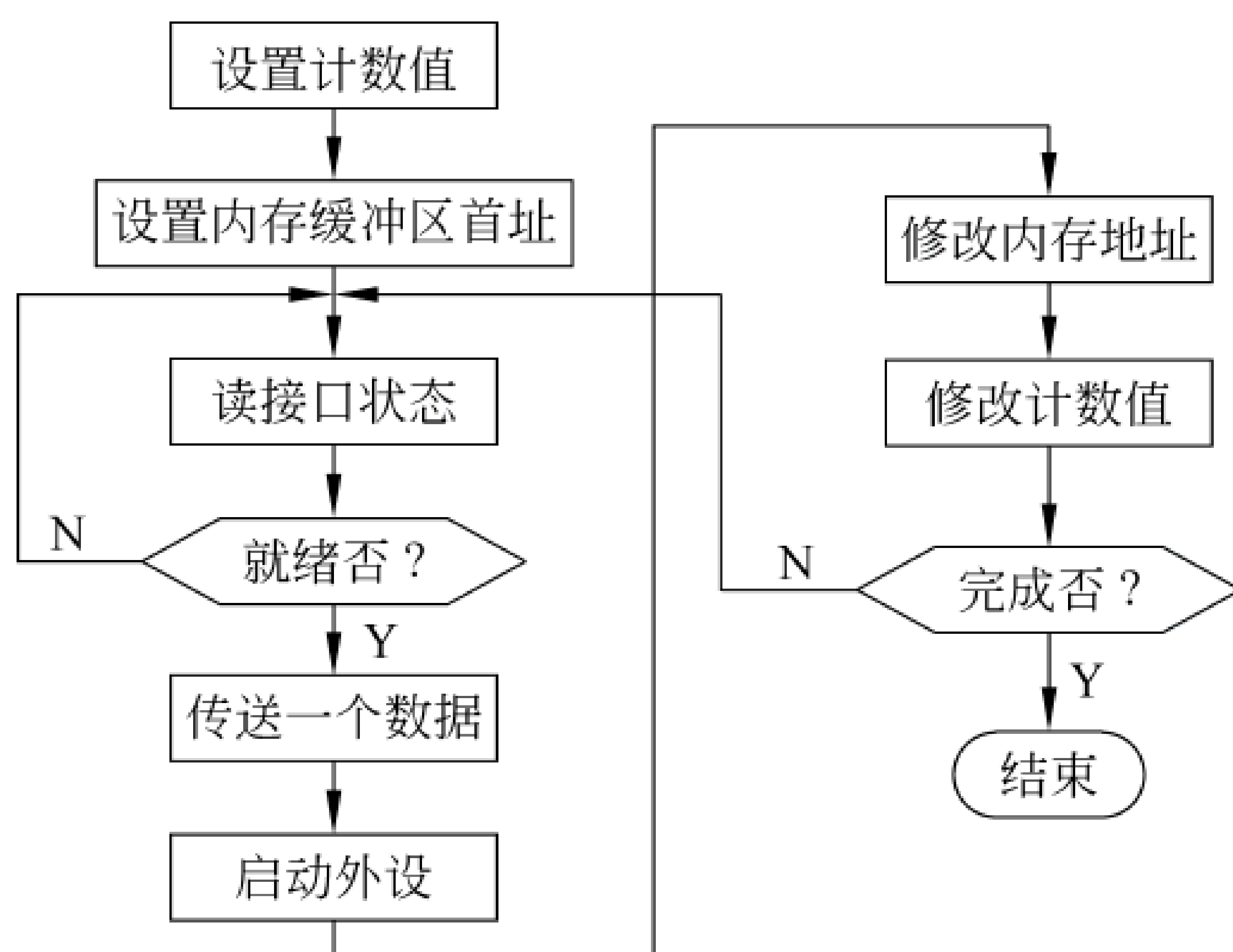


图9.35 独占查询程序流程图

从图 9.35 中可看出，在任何一个数据传送之前，必须先读接口的状态，判断接口是否“就绪”，只有在接口“就绪”的情况下，才继续进行传送。否则，CPU 将一直处于等待状态直到外设完成任务而使接口满足条件为止。这里“就绪”的含义，对于输入设备而言，意味着设备已将数据送入接口中的数据缓冲器，CPU 可以从接口取数据；对于输出设备而言，意味着数据缓冲器已空，CPU 可以将新的数据送到接口中。

例如，打印机的输出控制方式可采用这种查询方式，首先主机向打印控制器输出一个控制字符，然后就去读打印控制器中的“忙”状态位。如果“忙”，则主机等待；如果不忙，主机发送选通信号，把打印字符送打印缓冲器。输出满一行字符后，主机发回车或换行命令，打印机才真正开始打印。

程序查询 I/O 方式的特点是简单、易控制、外围接口控制逻辑少。但是，CPU 需要从外设接口读取状态并在条件不满足时继续等待外设完成任务，由于外设的速度比处理器慢得多，所以，在 CPU 等待外设完成任务的过程中浪费了许多处理器时间。

9.5.2 程序中断 I/O 方式

在计算机发展进程中，处理器速度提高很快，而外设速度改善较慢，两者之间速度相差很大，在独占程序查询方式中，CPU 和外设采用完全串行的工作方式，使处理器大量宝贵时间花在等待极慢速的外设上。为避免 CPU 长时间等待外设，提出了“中断”控制 I/O 方式。

1. 中断的概念

本教材第 6 章的第 6.5 节曾经介绍过“异常和中断”的概念，并说明“中断”一词在不同系统和不同教科书中有其不同的内涵，在本教材中，“中断”就是指外部设备向 CPU 发出的外部中断。

中断控制 I/O 方式的基本思想是，当 CPU 需要进行一个 I/O 操作时，先启动外设工作，并挂起执行 I/O 操作的进程，然后从就绪队列中选择另一个进程执行，此时，外设和 CPU 并行工作。当外设完成操作，便向 CPU 发中断请求。CPU 响应请求后，就中止现行程序的执行，转入一个“中断服务程序”，在“中断服务程序”中完成数据传送任务，并启动外设进行下一个操作。“中断服务程序”执行完后，返回原被中止的程序断点处继续执行。此时，外设和 CPU 又开始并行工作。上述过程可由图 9.36 和图 9.37 来说明。

中断方式下，除了可以实现外设和 CPU 并行工作外，还能处理一些外设的异常事件和特殊事件。例如，打印机缺纸、磁盘寻道结束、DMA 传送结束等。与前面第 6 章介绍的“内部异常”一样，计算机系统能及时捕捉到外部设备的这些异常或特定事件，并及时予以处理。

现代计算机系统中都配有完善的异常和中断处理系统，CPU 的数据通路中有相应的异常和中断的检测和响应逻辑，在外设接口中有相应的中断请求和控制逻辑，操作系统中有相应的中断服务程序。这些异常和中断处理硬件线路和中断服务程序有机结合，共同完成异常和中断的处理过程。中断 I/O 方式通过让处理器执行相应的中断服务程序来完成输入输出的任务，所以称为程序中断 I/O 方式。

外部中断由外设完成任务或出现特殊情况引起。如外设任务完成、打印机缺纸、磁盘检验错、采样计时到、键盘输入等。它和前面第 6 章介绍的内部异常在本质上是一样的。但是，两者相比它们有两个重要的不同点。

(1) “缺页”或“溢出”等异常事件是由特定指令在执行过程中产生的，而中断相对于指

令的执行则是异步的。也就是说，中断不和任何指令相关联，也不阻止任何指令的完成。因此，CPU 只需要在开始一个新指令之前检测是否有外部发来的中断请求即可。

(2) 异常的发生和异常事件的类型是由 CPU 自身发现和识别的，不必通过外部的某个信号通知 CPU，而对于中断，CPU 必须通过对外部中断请求线进行采样，并从总线上获取相应的中断源设备的标识信息，才能获知哪个设备发生了何种中断。

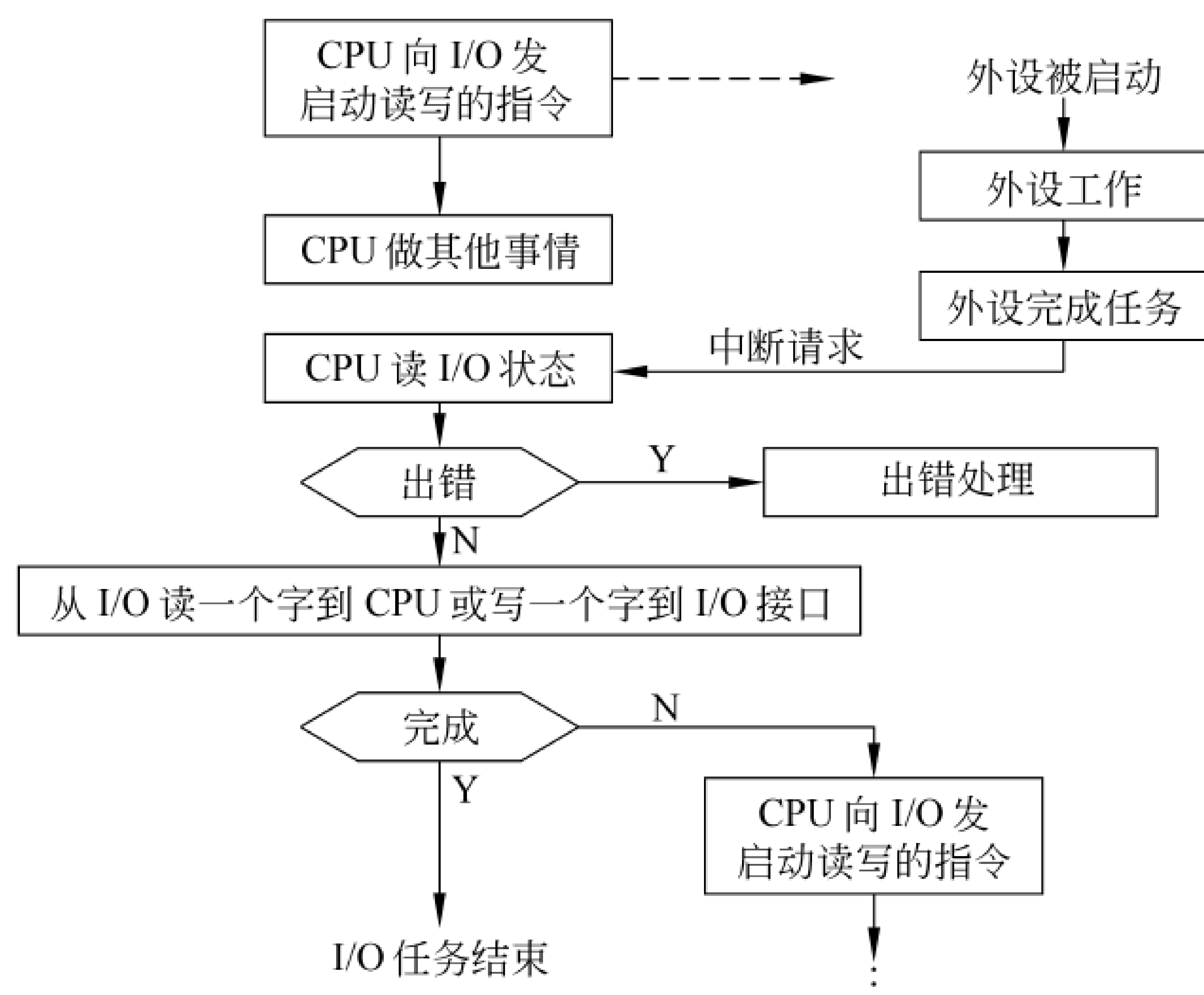


图 9.36　中断控制 I/O 过程

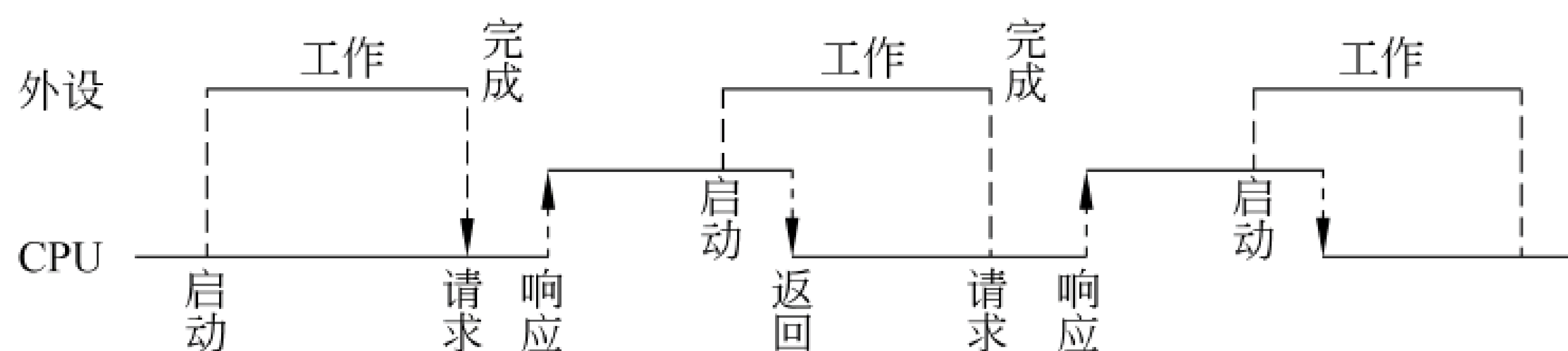


图 9.37　CPU 与外设并行工作

2. 中断系统的基本职能和结构

现代计算机的中断处理功能相当丰富，没有配置中断系统的计算机是令人无法想象的。每个计算机系统的中断功能可能不完全相同，但其基本功能不外乎以下几个方面。

(1) 及时记录各种中断请求信号。对于外部中断，中断系统中必须要有能够及时记录各种外部中断请求信号的部件，一般是用一个中断请求寄存器来保存。

(2) 自动响应中断请求。中断事件是在外部设备需要 CPU 干预时产生的，所以 CPU 必须能够在发生中断事件后，自动响应并处理。中断响应是在 CPU 执行指令的流程中固定安排的，总是在一条指令执行完、取出下条指令前去检查有无中断请求发生。若有，则根据情况决定是否响应和响应哪个中断请求。

(3) 自动判优。在计算机系统中，中断源有很多，中断被响应前，可能会有多个中断源提出中断请求。因此，中断系统中必须要有相应的中断判优逻辑，在有多个中断请求同时产生时，能够判断出哪个中断的优先级高，选择优先级高的中断先被响应。

(4) 保护被中断程序的断点和现场。因为中断响应后要转去执行中断服务程序，而执

行完中断服务程序后，还要回到原来的程序继续运行。所以原程序被中止处的指令地址和原程序的程序状态和各寄存器的内容等必须被保存，以便能正确回到原被中止处继续执行。

(5) 中断屏蔽。现代计算机大多采用中断嵌套技术。也就是说，中断系统允许 CPU 在执行某个中断服务程序时，被新的中断请求打断。但是并不是所有的中断处理都可被新的中断打断，对于一些重要的紧急事件的处理，就要设置成不可被其他新的中断事件打断，这就是中断屏蔽的概念。中断系统中要有中断屏蔽机制，使得每个中断可以设置它允许被哪些中断打断，不允许被哪些中断打断。这个功能主要通过在中断系统中设置中断屏蔽字来实现。屏蔽字中的每一位对应某一个外设或中断源，称为该外设的中断屏蔽位，例如，用“1”表示允许中断，“0”表示不允许中断（即屏蔽中断）。CPU 还可以通过在程序中执行相应的指令来修改屏蔽字的内容，从而动态地改变中断处理的先后次序。

中断系统的基本结构如图 9.38 所示。

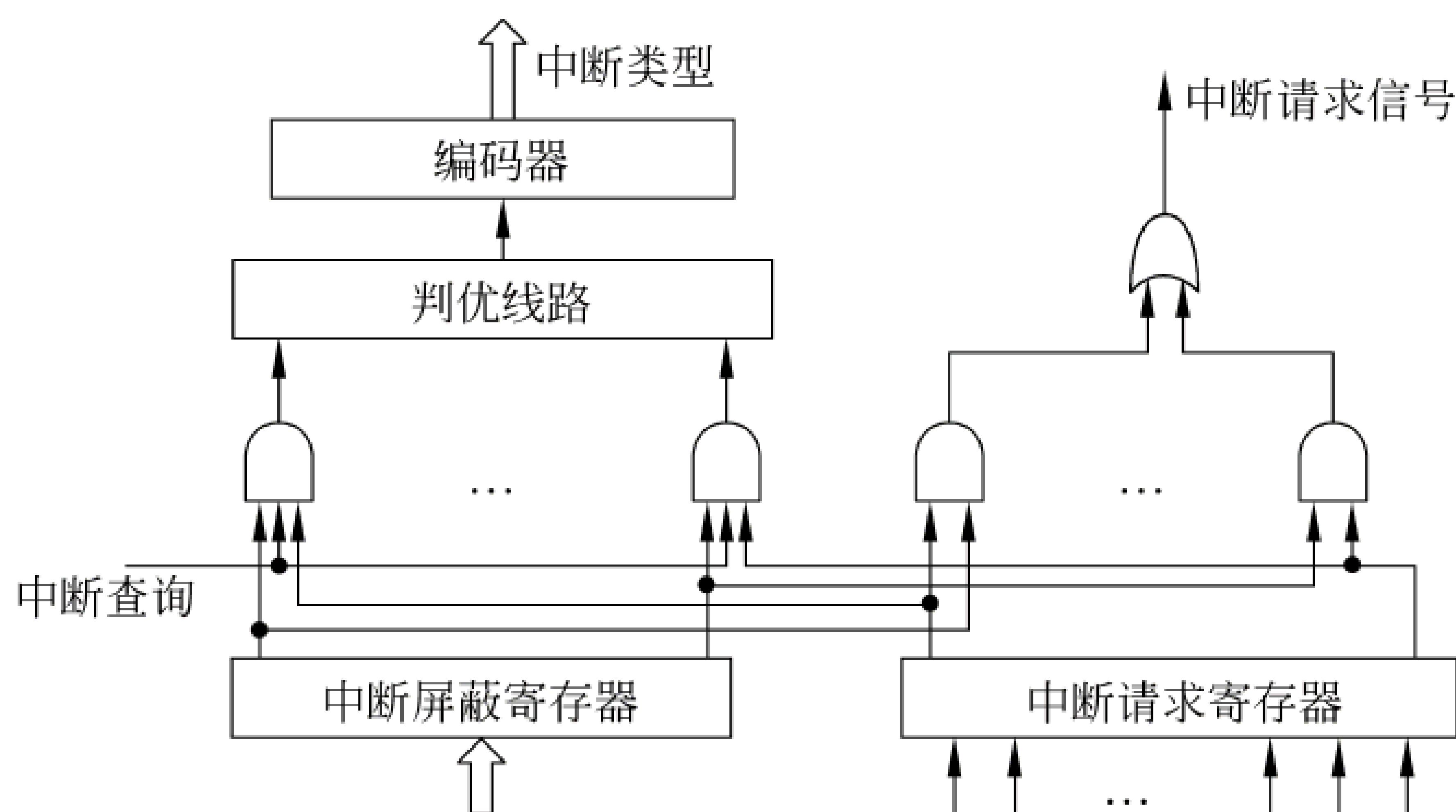


图 9.38　中断系统的基本结构

从图 9.38 可以看出，来自各个设备的外部中断请求记录在中断请求寄存器中的对应位，每个中断源有各自对应的中断屏蔽字，在进行相应的中断处理之前它被送到中断屏蔽寄存器中。在 CPU 运行程序时，每当 CPU 完成当前指令的执行、取出下一条指令之前，就会通过采样中断请求信号线来自动查看有无中断请求信号。若有，则会发出一个相应的中断回答信号，启动图 9.38 中的“中断查询”线，在该信号线的作用下，所有未被屏蔽的中断请求信号一起送到一个判优线路中，判优线路根据中断响应优先级选择一个优先级最高的中断源，然后用一个编码器对该中断源进行编码，得到对应的中断源设备类型号（即中断源的标识信息，称为中断类型）。CPU 取得中断源的标识信息后，经过一系列相应的转换，就可得到对应的中断服务程序的首地址，在下一个指令周期开始，CPU 执行相应的中断服务程序。

在中断处理（即执行中断服务程序）过程中，若又有新的优先级更高的中断请求发生，那么 CPU 应立即中止正在执行的中断服务程序，转去处理新的中断，这种情况被称为多重中断或中断嵌套，如图 9.39 所示。

图 9.39 中，假定在执行用户程序时，发生了 1＃中断请求，因为用户程序不屏蔽任何中断，所以就响应 1＃中断，将原程序的断点保存在堆栈中，然后调出 1＃中断服务程序执行，而在执行 1＃中断的过程中，又发生了 2＃中断，而 2＃中断的处理优先级比 1＃高，也即 1＃中断的屏蔽字对 2＃中断是开放的，此时，就中止 1＃中断的处理，响应 2＃中断，把 1＃中断的断点信息保存在堆栈中，调出 2＃中断的中断服务程序执行。同样，3＃中断也可以打断 2＃中断的执行。当 3＃中断处理完返回时，系统从栈顶取出返回的断点信息，这样，从 3＃中

断返回后，首先回到2＃中断的断点（K3＋1）处，而不是回到1＃中断或原主程序执行。利用堆栈能正确地实现中断嵌套。

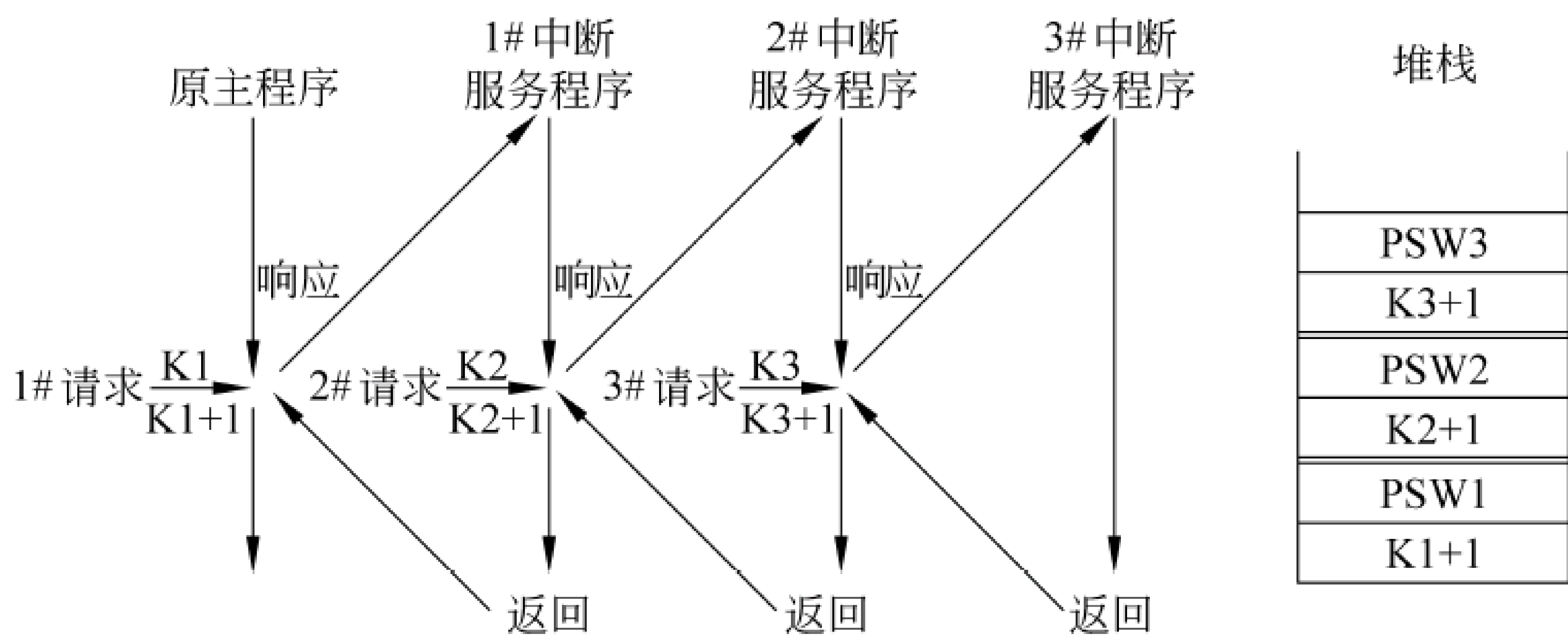


图 9.39　中断嵌套过程

从上面描述的过程来看，中断系统中存在两种中断优先级。一种是中断响应优先级，另一种是中断处理优先级。中断响应优先级是由查询程序或硬件判优排队线路决定的优先权，它反映的是多个中断同时请求时选择哪个先被响应。中断处理优先级是由各自的中断屏蔽字来动态设定的，反映了本中断与其他所有中断之间的处理优先关系。在多重中断系统中通常用中断屏蔽字对中断处理优先权进行动态分配。

3. 中断过程

中断过程包括两个阶段：中断响应和中断处理。中断响应阶段由硬件实现，而中断处理阶段则由CPU执行中断服务程序来完成，所以中断处理是由软件实现的。

（1）中断响应

中断响应是指主机发现中断请求，中止现行程序的执行，到调出中断服务程序这一过程，因此，中断响应过程是处理器从一个进程切换到另一个进程的过程。在此过程中处理器应完成以下三个任务。

① 保存好被中断程序断点处的关键性信息

为保证被中断程序在从中断服务程序返回后能在断点处继续正确执行下去，有两类信息不能被中断服务程序破坏。一类是用户可见的工作寄存器的内容，这些工作寄存器存放着程序执行到断点处的现行值，一般把这类信息称为现场；另一类是指令不可访问的程序计数器PC和程序状态字寄存器PSWR的内容，通常称作断点信息。对于现场信息，因为是用户可访问的，所以通常在中断服务程序中通过指令把它们保存到一个特定的存储区（如堆栈），即由软件实现；对于断点信息，因为必须将中断服务程序的首地址装入到PC中才能转到中断服务程序执行，所以，原来在PC中的断点信息应在中断响应开始时由硬件自动保存到一个特定的地方（堆栈或专门寄存器）。PSWR也由硬件保存，但是，若系统提供了访问PSWR的指令，也可由软件来保存。

② 识别中断源并判优

中断响应的结果是调出相应的中断服务程序来执行，因此，在中断响应过程中，处理器必须能够识别出哪些中断有请求，并且在有多个中断请求出现的情况下，选择响应优先级最高的中断。

③ 调出中断服务程序

处理器通过相应步骤得到所选择的中断源对应的中断服务程序的首地址和初始程序状

态字，并把它们分别送 PC 和 PSWR。这样，在中断响应结束后的下一个时钟周期，处理器就转入相应的中断服务程序执行。

处理器响应中断的时间越短越好。中断响应时间是中断系统设计时需考虑的一个重要指标，它反映了计算机系统的灵敏度。显然，中断响应时间与断点保存的时间、中断源识别和判优的速度以及获得中断服务程序首地址和初始状态的时间都有关系。不同的中断响应处理机制，得到的中断响应时间不同。例如，MIPS 处理器采用一种简单的响应机制，断点信息 PC 直接保存在特殊的寄存器 EPC 中，而不需要访问内存来存取堆栈，中断源的识别和判优处理也由软件进行，处理器只要直接把进行中断源识别和判优处理的程序首地址送到 PC 即可完成中断响应过程。因此，MIPS 处理器的中断响应时间很短，但由于是软件查询中断源，所以转到具体中断处理的时间较长。

很显然，在保护断点和现场的过程中，如果又有新的中断被响应，则原被保存的断点或现场就会被破坏，因而，需要有一种机制能保证断点和现场的保护过程不被新的中断请求打断。通常，用一个“中断允许”触发器（或状态位）来实现控制。第 6 章 6.5.2 节中介绍过，当 CPU 中的“中断允许”触发器为 1 时，CPU 处于“中断允许”（或“开中断”）状态。CPU 只有在“中断允许”状态时，才有可能响应新的中断请求。

因此，中断响应的条件有以下三个：

① CPU 处于“开中断”状态。

② 至少要有一个未被屏蔽的中断请求。

③ 当前指令刚执行完（对于非流水线处理器，此时 PC 中存放的是下条指令的地址）。

当处理器同时满足上述三个条件时，就响应中断，进入中断响应周期，它是一种特殊的机器执行周期。在中断响应周期中，通过执行一条隐指令，完成以下几个操作：

① 关中断：将中断允许标志置为禁止（即“关中断”）状态。

② 保护断点：将 PC 和 PSW 送入堆栈或特殊寄存器。

③ 识别中断源并转中断服务程序：通过某种方式，获得优先级最高的中断源所对应的中断服务程序的首地址和初始 PSW，并分别送 PC 和 PSWR。

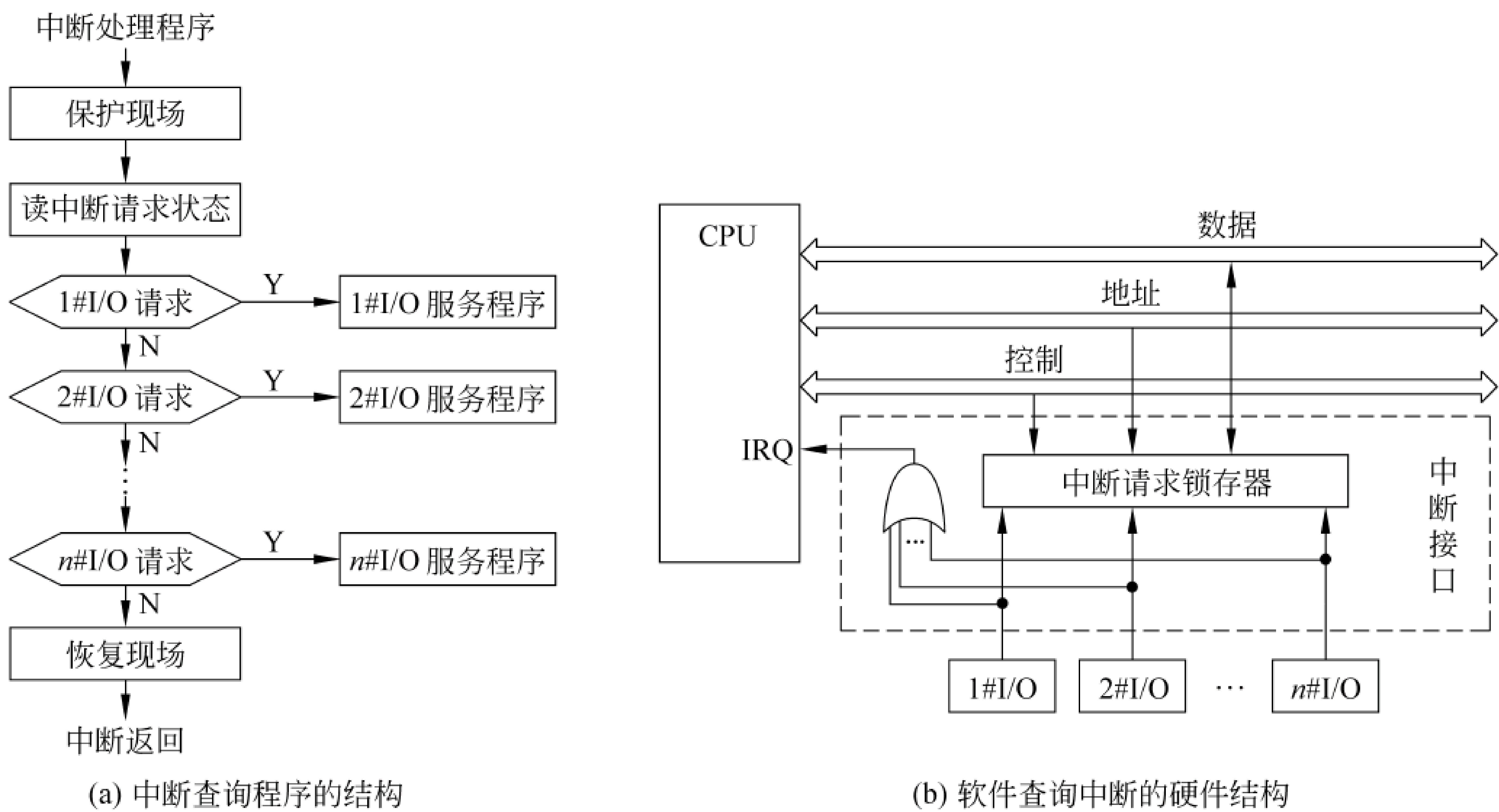
中断源的识别和判优方法可分为软件查询和硬件判优两大类。

① 软件查询方法

当 CPU 检测到中断请求时，通过中断响应，自动转到一个特定的中断查询程序，在中断查询程序中，按中断优先顺序依次查询哪个设备有中断请求，并转到第一个查询到有请求的中断服务程序去执行。这种软件中断识别方式的中断接口硬件结构很简单，只要一根中断请求线和一个中断请求寄存器，而且，可通过改变软件中的查询顺序来改变中断响应优先级，因而比较灵活。但是，因为软件查询速度慢，因而中断请求可能无法得到实时响应。图 9.40 给出了这种软件查询方式下中断查询程序的结构和中断接口的硬件结构。

② 硬件判优方法

硬件判优也称为向量中断方式，是一种不同于软件查询的中断处理技术。它通过中断接口中的判优线路和优先权编码器，得到所有未被屏蔽的中断请求中具有最高优先权的中断类型（中断源标识），从而找到对应中断服务程序的首地址 PC 和初始 PSW。



(a) 中断查询程序的结构　　(b) 软件查询中断的硬件结构

图 9.40　软件查询中断方式下的程序结构和硬件结构

通常把中断服务程序的首地址 PC 和初始 PSW 称为中断向量 IV(Interrupt Vector)。在向量中断方式下,所有中断向量存放在一个中断向量表(也称中断入口地址表)中,中断向量所在的地址称为向量地址 VA(Vector Address),如图 9.41 所示。因为每个中断向量在中断向量表中的位置可以用对应表项的号码来标识,如图 9.40 中的 1,2,…,n,所以,有些系统把中断向量表中与相应中断对应的表项编号称为中断向量,也有的处理器把它称为中断类型号。例如,如图 9.42 所示,8086/8088 的中断向量表位于主存区域 0000H～03FFH,共 256 个表项,每个表项占 4 个字节,记录对应中断服务程序的首地址 CS:IP。向量地址由中断类型号乘 4 得到。例如,除法错的中断类型号为 0,故其向量地址为 0;不可屏蔽中断 NMI 的中断类型号为 2,故其向量地址为 8。

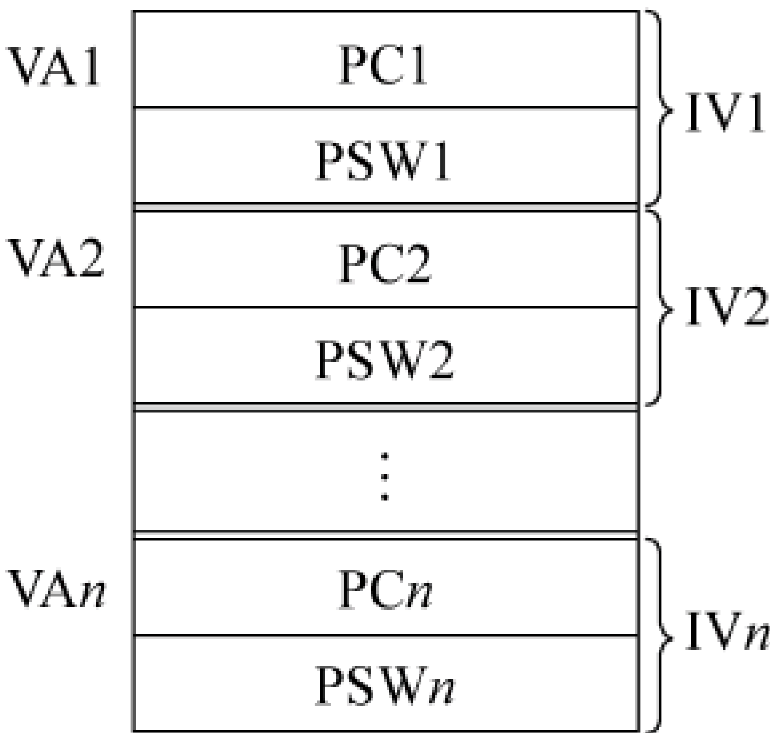


图 9.41　中断向量表

表项	地址
CS:IP　(除法错)	00~03
CS:IP　(单步)	04~07
CS:IP　(NMI)	08~0B
⋮	
CS:IP	
CS:IP	3FC~3FF

图 9.42　8086/8088 的中断向量表

根据判优电路实现方式的不同,硬件判优法有两种:链式查询和独立请求。软件查询方式由于用软件进行中断查询,所以响应速度慢。改进的一个途径就是将查询程序硬化,采用菊花链方式进行,也称为硬件轮询,其对应的中断请求和中断响应电路结构和过程类似于总线裁决中的链式查询。由于链式查询的优先级固定,中断响应速度慢,特别是由于无法为每个中断设置屏蔽字,所以也不支持多重中断,因此,现代计算机系统大多采用独立请求方

式来进行中断源的识别和判优。这种方式下，中断接口中有一个专门的中断控制器，在中断控制器中有相应的优先级分析器和编码器。各个中断请求信号和对应的中断屏蔽位进行“与”操作后，送到优先级分析器中进行排队判优，最后通过编码器输出对应的中断类型号，即中断源标识信息。这种方式下，中断响应速度快。如果是可编程的中断控制器，则优先级可灵活设置。Intel 8259A 就是一个典型的可编程中断控制器。图 9.43 给出了 8259A 的内部结构示意图。

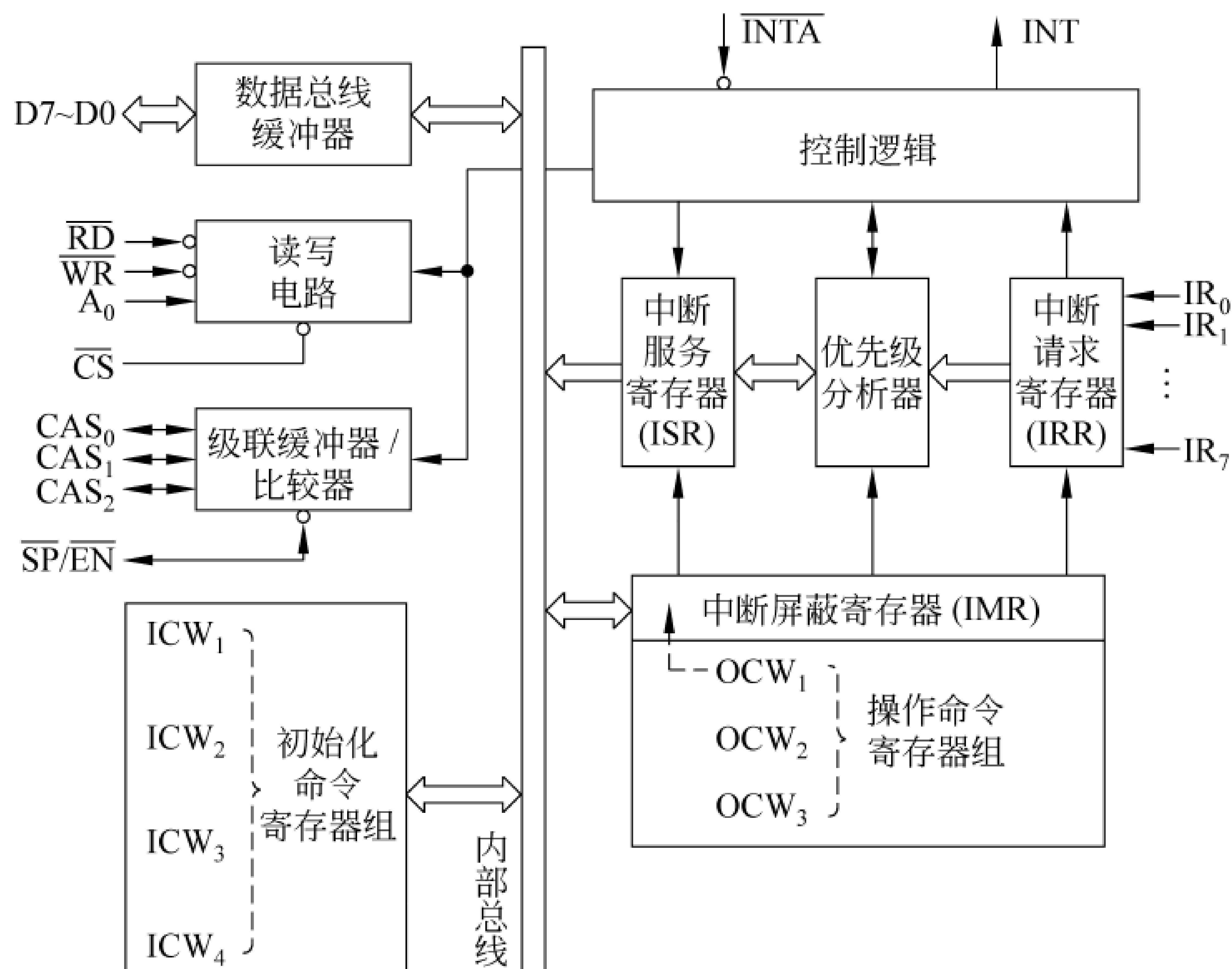


图 9.43　8259A 内部结构图

8259A 中断控制器的主要功能有：

① 中断请求锁存、中断屏蔽、中断优先级排队、中断源标识信息的生成等；

② 既可支持程序查询式中断，又可支持向量式中断；

③ 支持 8 级优先权，通过多片级联，最多可构成 64 级中断；

④ 各种中断功能可通过编程来设定和更改。

例如，8259A 通过 IR_0～IR_7 接收各路中断请求，存放于中断请求寄存器中，并汇集成一个公共的中断请求信号 INT，送往 CPU。CPU 响应中断后，发出中断响应信号$\overline{\text{INTA}}$回送给 8259A。优先级分析器经过中断优先级比较，把最高优先级的中断类型号经数据总线送给 CPU。在 CPU 内经简单变换，形成向量地址，据此访问中断向量表，取出中断入口地址后转相应中断服务程序执行。

一个 8259A 芯片能处理 8 个设备的中断请求，如果需要处理 8 个以上设备的中断请求，则可将若干个 8259A 芯片级联起来。使用级联方式，能处理多达 64 个设备的中断请求。

（2）中断处理

中断处理的过程就是 CPU 执行相应的中断服务程序的过程，不同的中断源其对应的中断服务程序不同。典型的中断处理分为三个阶段：先行段、本体段和结束段。

图 9.44 给出了中断服务程序的典型结构。

从图 9.44 中可以看出，在保存断点和现场的过程中，处理器处于“中断禁止”(“关中断”)状态，同样在恢复阶段也要让处理器关中断。如果在保存和恢复阶段处理器处在“开中断”状态的话，那么有可能在断点保存、现场保存和现场恢复的过程中，又响应了新的中断。这样，断点或现场就会被新中断破坏，因而不能回到原来的断点继续执行或因为现场被破坏而不能正确执行。

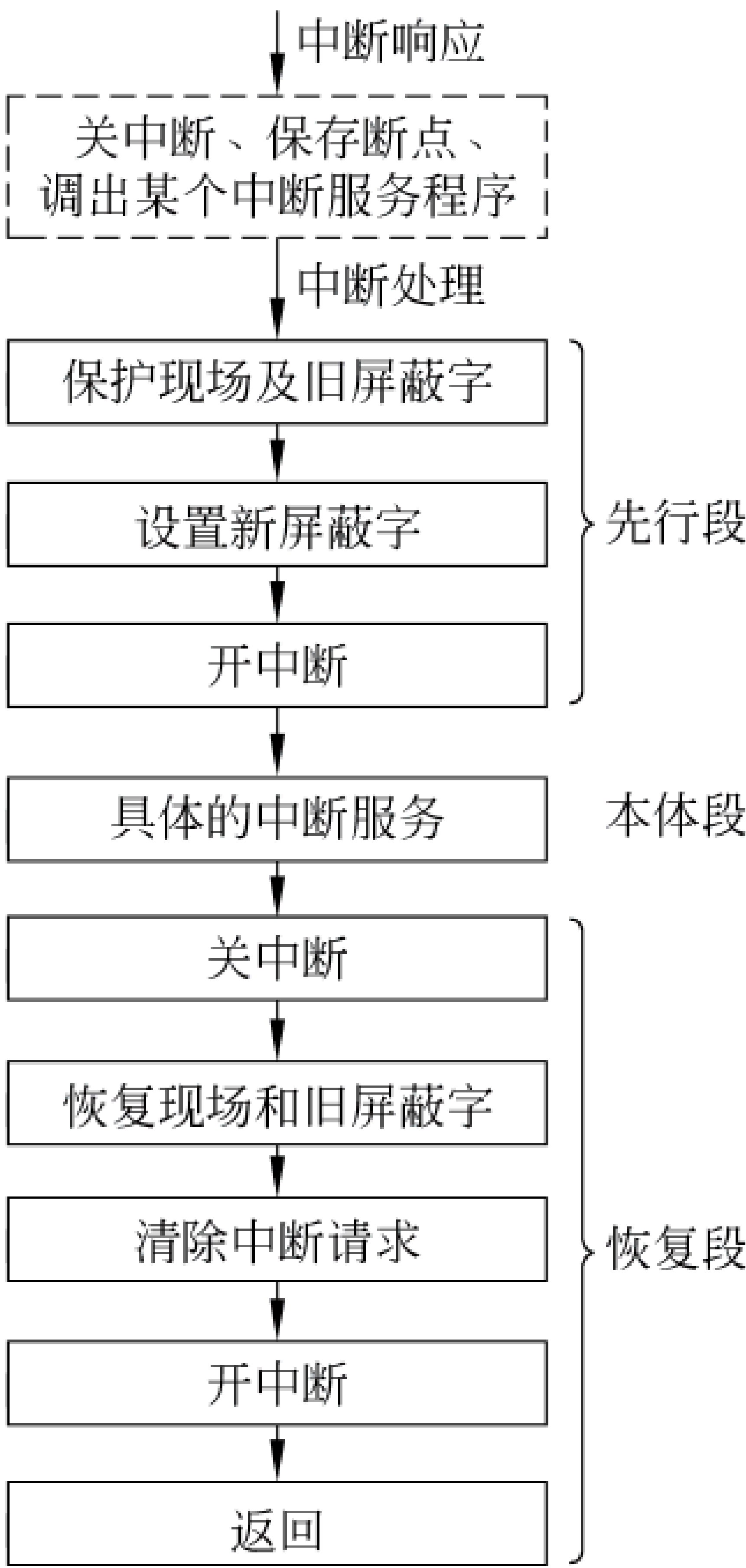


图 9.44　中断服务程序的典型结构

下面举例说明中断嵌套过程中处理器执行程序的过程。

例 9.2　假定某中断系统有 4 个中断源，其响应优先级为 1>2>3>4，分别写出处理优先级为 1>2>3>4 和 1>4>3>2 时各中断的屏蔽字以及 CPU 完成中断服务程序的过程。假定 CPU 在执行用户程序(主程序)时，同时发生了 1、3 和 4 级中断请求，而在执行 3 级中断服务程序的过程中又发生了 2 级中断请求。

解：中断处理优先级为 1>2>3>4 时的屏蔽字如表 9.2 所示。根据题意，在执行用户程序时同时发生了 1、3 和 4 级中断，由于响应优先级为 1>2>3>4，所以应先响应 1 级中断，1 级中断的屏蔽字为 1111，所以在 1 级中断处理过程中不响应任何中断，直到 1 级中断处理完回到用户程序。此时，还有 3 和 4 级中断未被处理，根据响应优先级，应先响应 3 级中断，调出 3 级中断服务程序执行，在处理 3 级中断的过程中，出现了 2 级中断请求，因为 3 级中断的屏蔽字为 0011，即对 2 级中断是开放的，所以 2 级中断被响应，打断了 3 级中断的处理。2 级中断处理过程中没有出现比它的处理优先级更高的中断请求，所以 2 级中断能一直处理完，然后回到被打断的 3 级中断服务程序继续执行，执行完后回到用户程序，最后响应 4 级中断，处理完 4 级中断后，再回到用户程序。其 CPU 执行程序的过程如图 9.45 所示。

表 9.2　中断处理优先级为 1>2>3>4 时的屏蔽字

中断程序级别	屏蔽字			
	1 级	2 级	3 级	4 级
第 1 级	1	1	1	1
第 2 级	0	1	1	1
第 3 级	0	0	1	1
第 4 级	0	0	0	1

假定 1 是屏蔽，0 是开放。

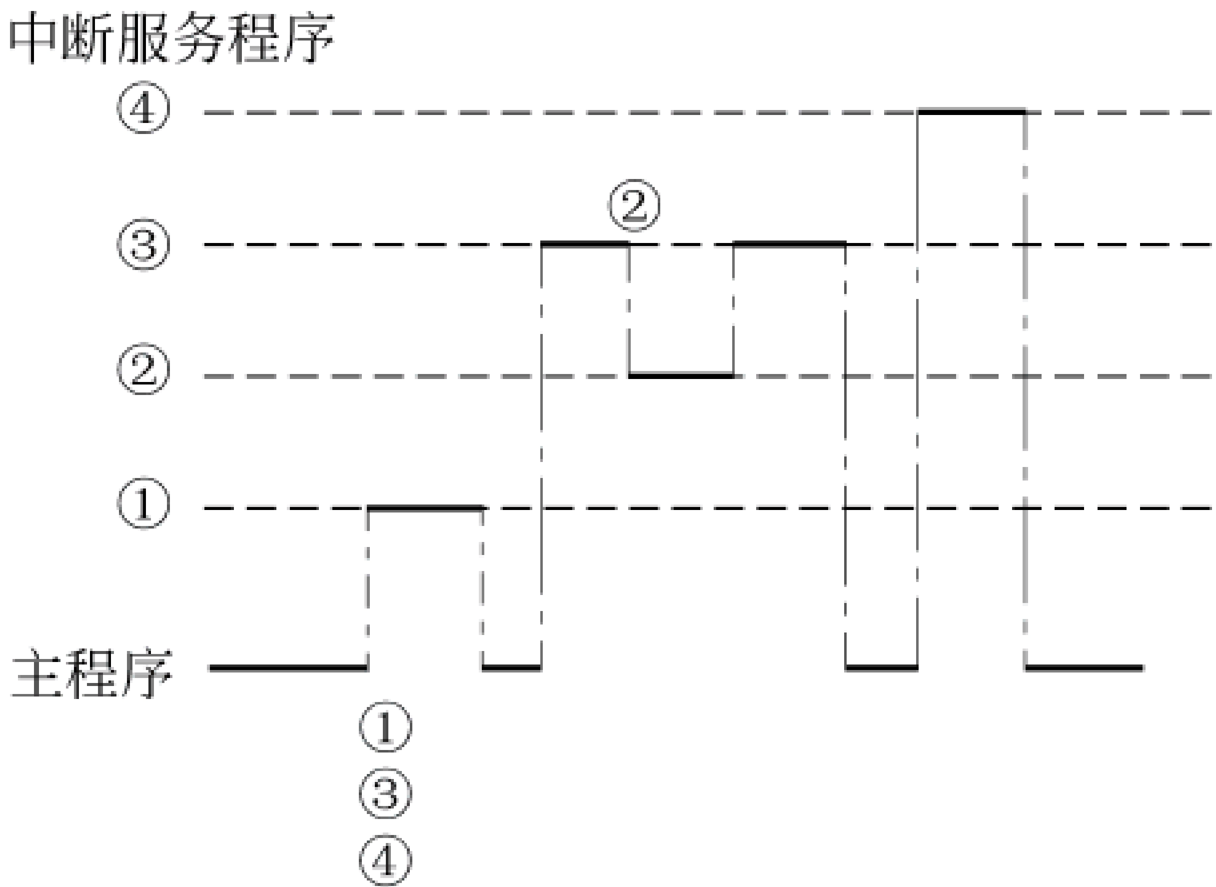


图 9.45　处理优先级为 1>2>3>4 时 CPU 运动轨迹

表 9.3 给出了中断处理优先级为 1＞4＞3＞2 时的屏蔽字。根据表 9.3 给定的屏蔽字，来考察 CPU 执行程序的运动轨迹。在执行用户程序时同时发生 1、3 和 4 级中断，根据响应优先级，应先响应 1 级中断，而 1 级中断的屏蔽字为 1111，所以在执行 1 级中断的过程中，不响应任何中断，直到 1 级中断处理完回到用户程序。此时，还有 3 和 4 级中断未被处理，根据响应优先级，应先响应 3 级中断，调出 3 级中断服务程序执行。在处理 3 级中断的过程中，出现了 2 级中断请求，同时在执行用户程序时发生的 4 级中断请求还未被响应，因为 3 级中断的屏蔽字为 0110，即对 2 级中断屏蔽而对 4 级中断开放，此时，响应优先级排队电路中，只有 4 级中断请求被排队判优，显然，4 级中断请求被响应，因此，CPU 中止 3 级中断，调出 4 级中断来处理。4 级中断处理过程中没有出现比它处理优先级更高的中断请求，因此，一直到它处理完，再回到被打断的 3 级中断。CPU 继续执行 3 级中断服务程序，执行完后，回到用户程序。最后响应 2 级中断，处理完 2 级中断后，再次回到用户程序。其 CPU 执行程序的轨迹如图 9.46 所示。

表 9.3　中断处理优先级为 1＞4＞3＞2 时的屏蔽字

中断程序级别	屏 蔽 字			
	1 级	2 级	3 级	4 级
第 1 级	1	1	1	1
第 2 级	0	1	0	0
第 3 级	0	1	1	0
第 4 级	0	1	1	1

假定 1 是屏蔽，0 是开放。

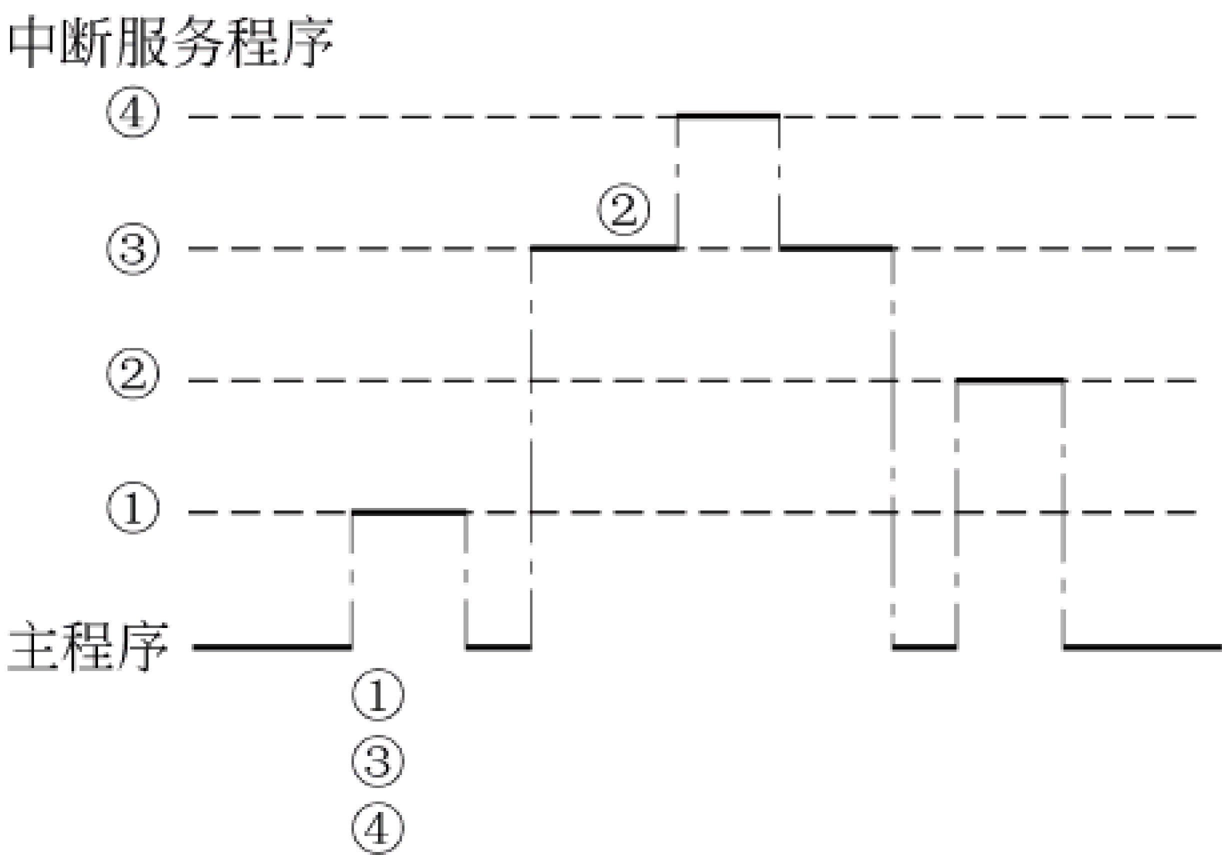


图 9.46　处理优先级为 1＞4＞3＞2 时 CPU 运动轨迹

前面给出了在外设和主机之间进行数据交换的两种输入输出方式：程序查询和程序中断方式。这两种方法用在低带宽设备上较好，因为对于低速设备，我们更关心怎样降低设备控制器和接口的成本，而不是关心怎样提高数据传输速率。下面的例子说明用中断方式控制硬盘和主机之间的数据交换时 CPU 的开销如何。

例 9.3　假定一个字长为 32 位的 CPU 的主频为 500MHz，即 CPU 每秒钟产生 500×10^6 个时钟周期。硬盘使用中断 I/O 方式进行数据传送，其传输速率为 4MBps，每次中断传输一个 16 字节的数据，要求没有任何数据传输被错过。每次中断的开销（包括用于中断响应和中断处理的时间）是 500 个时钟周期。如果硬盘仅有 5% 的时间进行数据传送，那么，CPU 用于硬盘数据传送的时间占整个 CPU 时间的百分比为多少？

解：若硬盘数据传送采用中断 I/O 方式，则每次中断传输 16 字节。为保证没有任何数据传输被错过，CPU 每秒钟应该至少执行 4MB/16B＝250k 次中断，因此，每秒钟内用于硬盘数据传输的时钟周期数为 $250k\times500=125\times10^6$，故 CPU 用于硬盘数据传送的时间占 $125\times10^6/(500\times10^6)=25\%$。

从以下另外一个角度来考虑也可以得出同样的结论。由题意知，CPU 通过中断方式进行硬盘数据的传送，硬盘每准备好一个 16 字节的数据，则发出中断请求。硬盘准备好一个 4 字块数据（16B）的时间为 16B/4MB＝4/M 秒，CPU 用于一个数据输入输出的时间（包括

中断响应并处理的时间)是500个时钟周期,相当于500/500M=1/M秒。由此可见,假定硬盘一直在工作的话,则硬盘每隔4/M秒申请一次中断,每次中断CPU用1/M秒进行硬盘数据传送,因此,CPU用在硬盘数据传送的时间占整个CPU时间的1/4,即25%。

假定硬盘并不是一直在操作,而是仅有5%的时间在工作,则CPU用于硬盘数据传送的时间占整个CPU时间的百分比为25%×5%=1.25%。

对于程序查询方式,在外设准备数据时,由于CPU一直在等待外设完成,所以CPU是有开销的,对于中断I/O方式,在外设准备数据时,CPU被安排执行其他程序,外设和CPU并行工作,因而CPU在外设准备数据时没有开销,只有响应和处理中断来进行数据传送时CPU才需要花费时间为I/O服务。这就是中断I/O方式相对于程序查询方式的优点。

但是,对于硬盘这种高速外设的数据传送,如果还是用中断I/O方式的话,则CPU用于I/O的开销是无法忽视的。高速外设速度快,因而中断请求频率高,导致CPU被频繁地打断,而且,由于需要保存断点和现场、开中断/关中断、设置中断屏蔽字等,使得中断响应和中断处理的额外开销很大,因此,在高速外设情况下,采用中断I/O方式传送数据是不合适的。

对于高速外设的数据传送,通常采用DMA方式。这种I/O方式能把CPU的负担减下来,在数据传送时,设备直接与存储器交换数据而不需要CPU的参与。

9.5.3 DMA方式

DMA(Direct Memory Access)称为直接存储器存取,该输入输出方式用专门的DMA接口硬件来控制外设与主存间的直接数据交换,数据不通过CPU。通常把专门用来控制总线进行DMA传送的接口硬件称为DMA控制器。在进行DMA传送时,CPU让出总线控制权,由DMA控制器控制总线,通过“窃取”一个主存周期完成和主存之间的一次数据交换,或独占若干个主存周期完成一批数据的交换。

DMA方式主要用于磁盘等高速设备的数据传送。这类高速设备的记录方式多采用数据块组织方式,数据块之间有间隙,因而数据传输时数据块之间的时间间隔较长,而数据块内部数据间的传输时间间隔较短,因此,这类设备大多采用成批数据交换方式。

DMA方式与中断I/O方式一样,也是采用“请求-响应”方式,只是中断I/O方式请求的是处理器的时间,而DMA方式下请求的是总线控制权。不过,在用DMA方式进行硬盘等高速设备的数据传输过程中,也要用到程序查询和中断I/O方式。图9.47给出了在磁盘和主存之间进行数据交换的过程示意图。首先,采用程序查询方式设置传送参数,通过执行相应的指令对有关参数寄存器进行初始化;然后CPU执行输出指令发出“寻道”命令到磁盘控制器,由磁盘控制器控制磁盘驱动器开始移动磁头,同时,CPU切换到其他程序执行;当磁盘完成寻道操作后,向CPU发出“寻道结束”中断请求;CPU响应并处理该中断请求,通过执行输出指令发出“查找扇区”命令到磁盘控制器,CPU启动“查找扇区”命令后,返回原程序

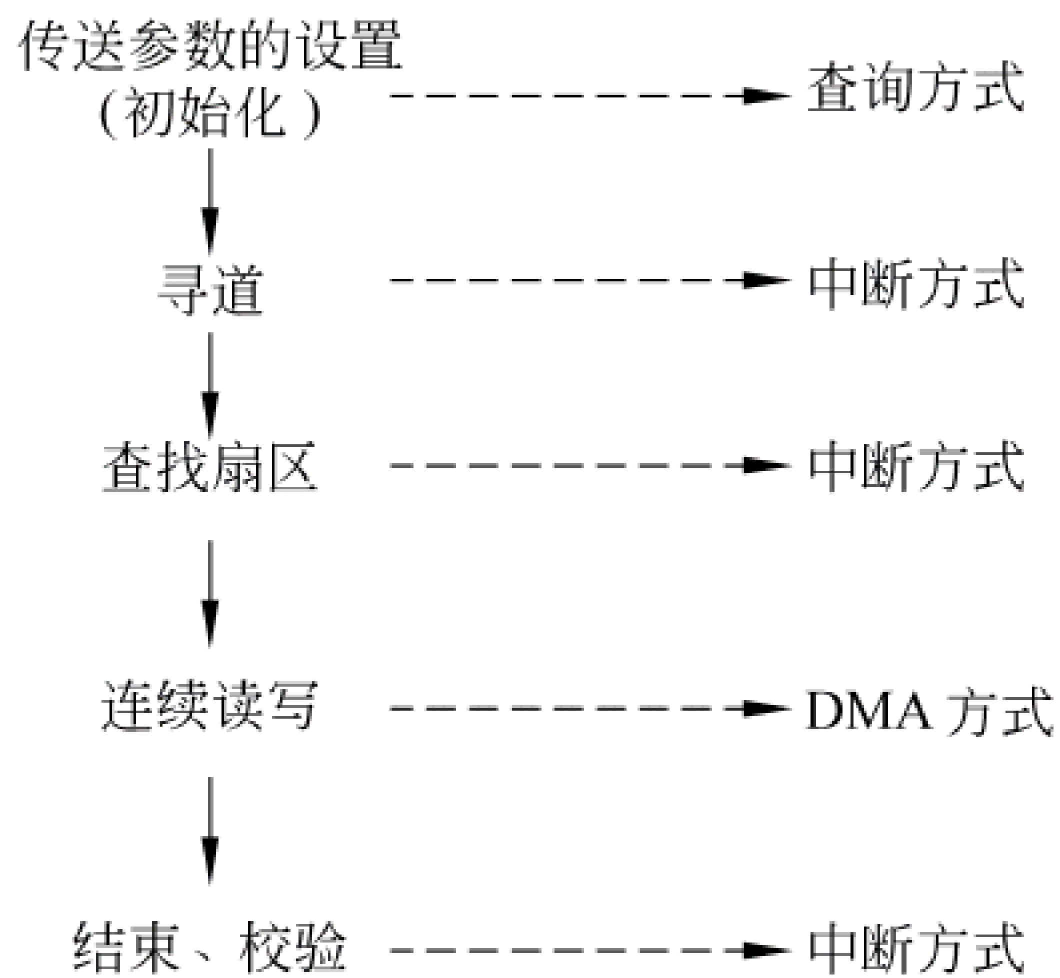


图9.47 DMA方式进行磁盘数据传送过程

执行；当磁盘完成扇区查找操作，就向 CPU 发出“查找结束”中断请求；CPU 响应并处理该中断，启动进行 DMA 传送，由 DMA 控制器控制数据在主存和磁盘之间进行数据传送；CPU 启动 DMA 传送后，又回到原程序继续执行，直到 DMA 传送结束；此时，由 DMA 控制器发出“DMA 结束”中断请求，CPU 响应并处理该中断请求，对传送的数据进行校验等后处理。

从上述过程来看，DMA 方式与程序查询和中断方式不同，CPU 不直接执行输入输出指令来实现数据传送，而只是进行一些辅助工作，包括传送参数的设置、各种外设操作的启动和数据校验等。这些辅助工作通过程序查询或中断 I/O 方式实现。

1. 三种 DMA 方式

由于 DMA 控制器和 CPU 共享主存，所以可能出现两者争用主存的现象，为使两者协调使用主存，DMA 通常采用以下三种方式之一进行数据传送。

（1）CPU 停止法。DMA 传输时，由 DMA 控制器发一个停止信号给 CPU，使 CPU 脱离总线，停止访问主存，直到 DMA 传送一块数据结束。

（2）周期挪用法。DMA 传输时，CPU 让出一个总线事务周期，由 DMA 控制器挪用一个主存周期来访问主存，传送完一个数据后立即释放总线。是一种单字传送方式。

（3）交替分时访问法。每个存储周期分成两个时间片，一个给 CPU，一个给 DMA，这样在每个存储周期内，CPU 和 DMA 都可访问存储器。

图 9.48 给出了以上三种方式下 CPU 和 DMA 访问主存的情况。

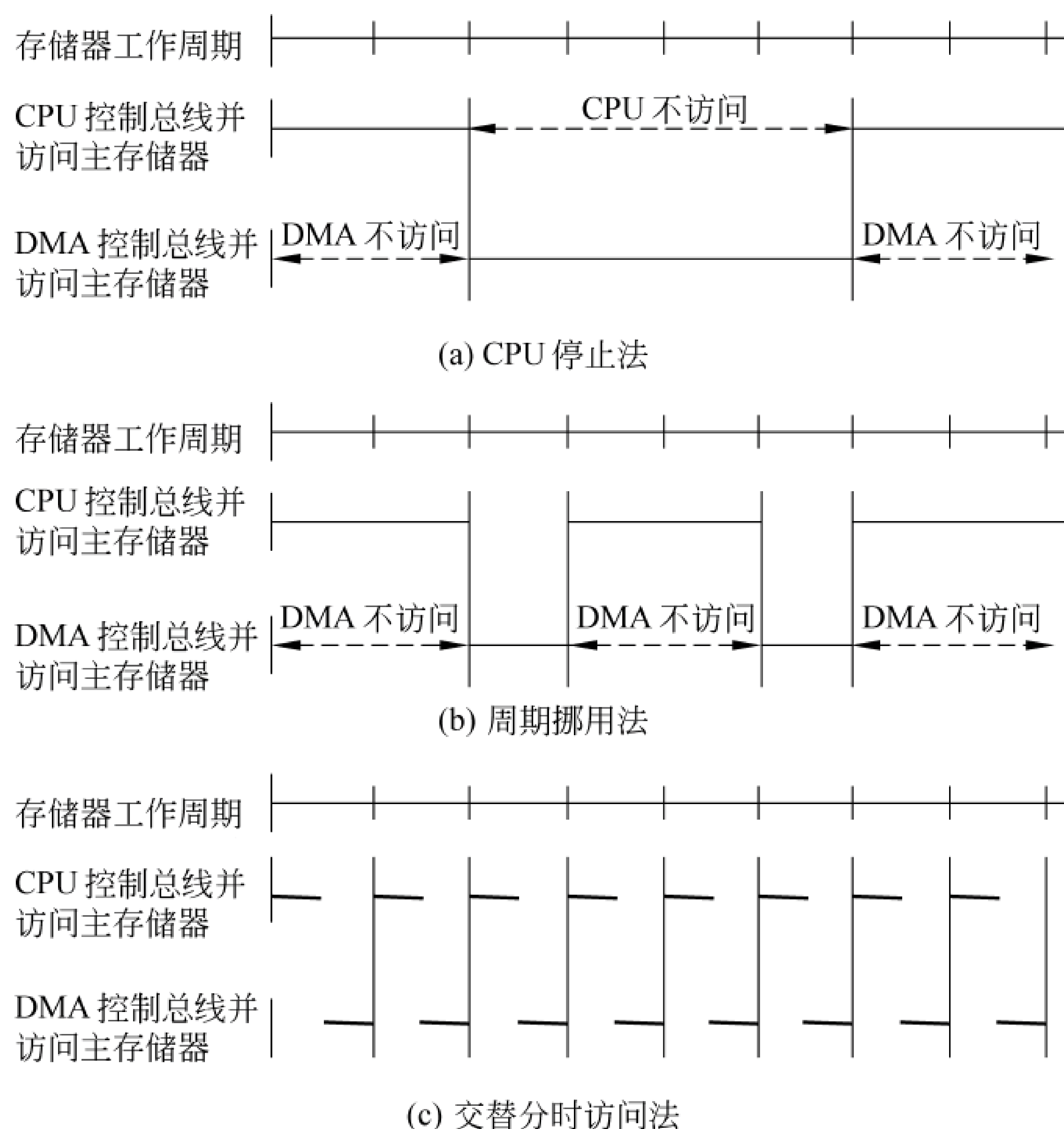


图 9.48　三种 DMA 方式

在图 9.48(a)所示的 CPU 停止法中，一旦 DMA 控制器占用了总线，它就一直控制总线直到所有数据传送完毕，因此在一个很长时间内，CPU 都不能使用总线，因而也就无法访问

主存,使 CPU 工作受到很大影响。即使是高速设备,两个数据之间的准备间隔时间也总大于一个主存周期,这使主存周期空闲程度比较高,主存周期未被充分利用。但这种方式的控制简单,在传输速率很高的外设中实现成组数据传送时比较适用。

弥补 CPU 停止法缺点的做法有两个。

(1) 在 DMA 接口中引入缓冲器。在 DMA 接口中采用一个小容量的半导体存储器,使外设先和这个小容量的高速存储器交换数据,然后再使用总线由小容量存储器与主存进行数据交换。这样可减少 DMA 传送数据时占用总线的时间,也就减少了 CPU 的等待时间。

(2) 采用周期挪用法。每次 DMA 传送完一个数据就释放总线,这样,在设备准备下一个数据时,CPU 能插空访问主存。

图 9.48(b)中给出周期挪用法的示意图。采用周期挪用法既能及时响应 I/O 请求,又能较好地发挥 CPU 和主存的效率。由于在下一个数据准备阶段,主存周期能被 CPU 充分利用,因此它适合于设备读写周期大于主存周期的情况。其缺点是每次 DMA 访存都要申请总线控制权和释放总线,增加了传输开销。

周期挪用方式下,外设要求进行 DMA 传送时可能会遇到以下三种情况之一。

(1) CPU 不需访问主存。例如,CPU 正在执行乘法指令,可能要花很长时间计算而不需马上访存,或者 CPU 访问 cache 命中。这时,CPU 在执行指令,总线可被 DMA 使用。这种情况下,CPU 和 DMA 不发生冲突,两者并行。

(2) CPU 正在访问主存。此时,必须等到主存存储周期结束后,CPU 让出总线,DMA 才能访存。

(3) CPU 也同时要访问主存。此时,CPU 和 DMA 出现访存冲突。因为不马上响应 DMA 请求的话,高速设备可能会发生数据丢失,所以,DMA 的总线优先权比 CPU 高。通常,先让 DMA 占用总线,窃取一个主存周期,完成一个数据交换后释放总线,因此,CPU 需延迟一段时间后才能访存。

图 9.48(c)给出了交替分时访问的情况,这种方式适用于 CPU 工作周期比主存存取周期更长的情况。在这种方式下,DMA 不需要总线使用权的申请和释放,CPU 既不停止主程序的运行也不进入等待状态,在 CPU 工作过程中,不知不觉地完成了 DMA 数据传送,故又被称为“透明的 DMA”方式。

2. DMA 接口的结构和功能

DMA 数据传送过程是由 DMA 接口的控制逻辑完成的,一般把 DMA 接口中控制传送的硬件逻辑称为 DMA 控制器,它能像 CPU 一样控制总线。当 CPU 要进行 I/O 时,CPU 就把要传送的数据个数、数据块在内存的首址、数据传送的方向(是读操作还是写操作)、设备的地址等参数送给 DMA 控制器,然后发送一个命令给 DMA 接口,启动外设进行数据准备工作。在这些工作完成后,CPU 就继续进行其他工作。而设备和主存交换数据的事情就交给了 DMA 控制器。DMA 控制器在需要的时候就申请总线控制权,占用总线完成设备和主存间的数据传送。在整个数据块传送过程中,CPU 完全不用管外设和 DMA 接口,一直在执行其他程序。直到 DMA 传送完毕,由 DMA 发来中断请求,CPU 才介入到该 I/O 操作中。图 9.49 是一个 DMA 接口的典型结构示意图。

如图 9.49 所示,DMA 接口中应有 I/O 传送所需的各种参数寄存器。包括主存地址寄存器、数据传送字计数器、控制寄存器、设备地址寄存器等,还要有相应的“DMA 请求”和

“总线请求”逻辑，以及总线控制逻辑。许多DMA控制器内还有一些数据缓冲存储器，以便在发生延迟传输时或等待成为总线主控设备期间，能够灵活地处理传送。

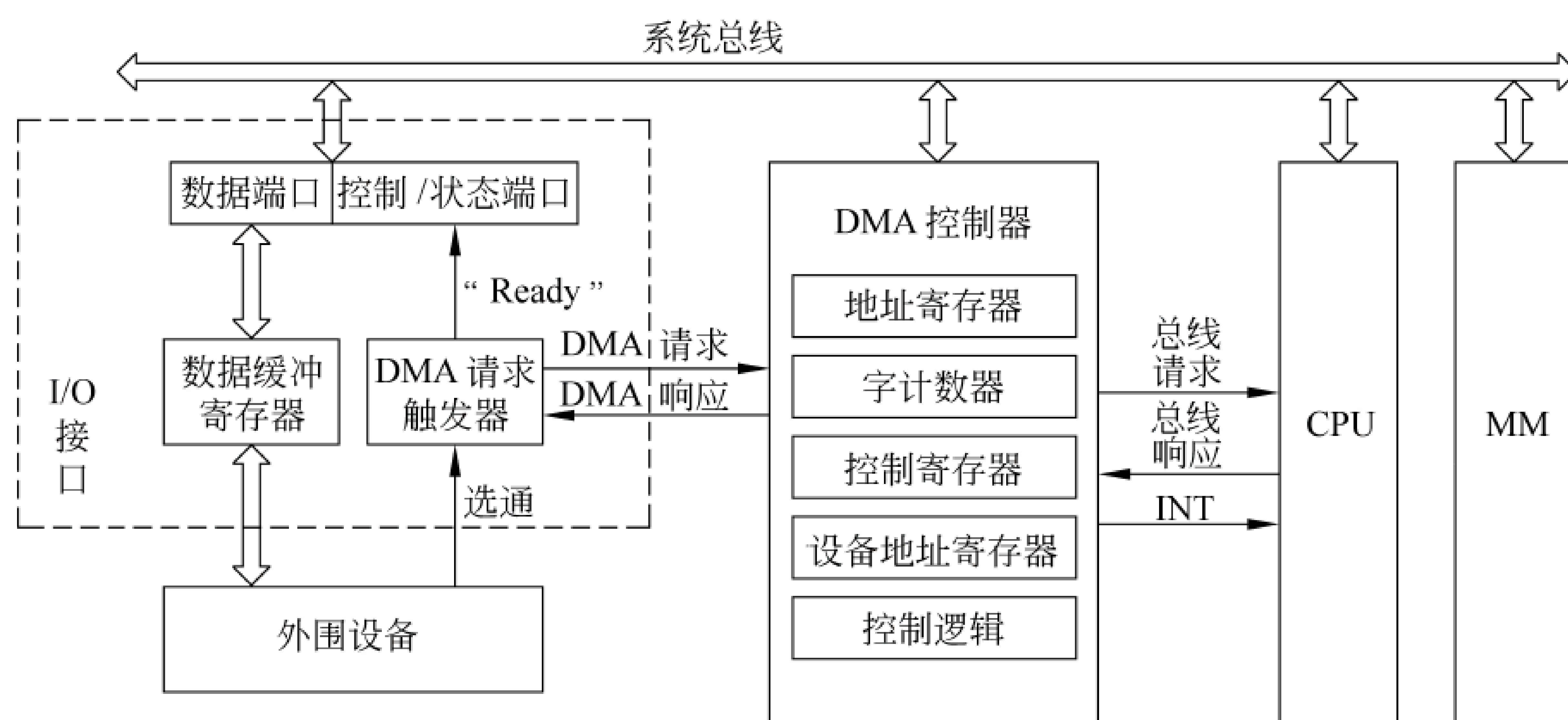


图9.49 DMA接口的结构

在一个计算机系统中，可能有多个DMA设备。图9.49中DMA接口由I/O设备的接口和DMA控制器组成。这是一种分离型的集中多路DMA接口。它将各设备接口中公用的DMA控制逻辑部分分离出来，成为通用的DMA控制器，而各设备有自己的I/O接口。DMA接口还可以采用合并型结构，将DMA控制逻辑和I/O接口合并为一个DMA接口。这种合并型结构又有选择型和单通道型两种，选择型DMA接口是各I/O接口的公共部分和DMA控制逻辑合并为一个DMA接口，通过I/O总线分时控制多个设备。例如，在一个具有单个处理器-主存总线和多I/O总线的系统中，每个I/O总线控制器通常含有一个DMA控制器，由这个DMA控制器处理任何在该I/O总线上的设备和存储器之间的数据传送。而单通道型DMA接口则是一个DMA接口对应一个设备。

不管是采用哪种结构的DMA接口，其主要功能有：

① 能接收外设发来的“DMA请求”信号，并能向CPU发“总线请求”信号；

② 当CPU发出“总线响应”信号后，能接管对总线的控制；

③ 能在地址线上给出主存地址，并自动修改主存地址；

④ 能识别传送方向以在控制线上给出正确的读写控制信息；

⑤ 能自动更新传送数据个数，并判断是否为0；

⑥ 能发出DMA结束信号，引起一次DMA结束中断，让CPU进行数据校验等后处理。

3. DMA操作步骤

DMA方式的I/O操作过程由以下几个步骤来完成。

第一步：DMA控制器的预置(初始化)

在进行数据传送之前，CPU先执行一段初始化程序，完成对DMA控制器中各参数寄存器的初始值的设定，主要操作包括以下三个方面。

(1) 准备内存区。若是从外设输入数据，则进行内存缓冲区的申请，并对缓冲区进行初始化；若是输出数据到外设，则先在内存准备好数据。

(2) 设置传送参数。执行输入输出指令来测试外设状态，并对DMA控制器设置各种

参数，例如：

内存首址→地址寄存器

字计数值→字计数器

传送方向→控制寄存器

设备地址→设备地址寄存器

(3) 启动外设进行相应的操作，然后 CPU 继续执行其他程序。

第二步：DMA 数据传送(DMA 传送)

CPU 对 DMA 传送参数进行预置并启动外设工作后，就把数据传送的工作交给了 DMA 控制器。整个传输过程中不需要 CPU 参与，完全由 DMA 硬件实现。对照图 9.49，其传送过程描述如下。

(1) 当外设准备好数据(从外设取数)，或准备好接收数据(向外设送数)时，就发“选通”信号，使数据送数据缓冲寄存器，同时 DMA 请求触发器置“1”。

(2) DMA 请求触发器向控制/状态端口发“Ready”信号，同时向 DMA 控制器发“DMA 请求”信号。

(3) DMA 控制器接受到“DMA 请求”信号后，就向 CPU 发“总线请求”信号。

(4) CPU 完成现行总线操作后，响应 DMA 请求，向 DMA 控制器发出“总线响应”信号。DMA 控制器接受到该信号后，向外设接口发 “DMA 响应”信号，使 DMA 请求触发器复位。此时，CPU 浮动它的总线，让出总线控制权，由 DMA 控制器控制总线。

(5) DMA 控制器给出内存地址，并在其读写线上发出“读”命令或“写”命令，随后在数据总线上给出数据。

(6) 根据读写命令，将数据总线上的数据写入存储器中，或写入外设数据端口，并进行主存地址增量，字计数值减 1。

若采用“CPU 停止法”，则循环第(6)步，直到计数值为“0”；若采用“周期挪用法”，则释放总线，下次数据传送时再按过程(1)到(6)进行。

第三步：DMA 结束处理(后处理)

根据计数值为“0”，发出“DMA 结束”信号送 DMA 接口，控制产生“DMA 结束”中断请求信号给 CPU，转入中断服务程序，做一些数据校验等后处理工作。

下面举例说明使用 DMA 方式进行硬盘和主存之间的数据传送时处理器所花的开销。

例 9.4 假定 CPU 的主频为 500MHz，即 CPU 每秒钟产生 500×10^6 个时钟周期。硬盘采用 DMA 方式进行数据传送，其数据传输率为 4MBps，每次 DMA 传输的数据量为 8KB，要求没有任何数据传输被错过。如果 CPU 在 DMA 初始化设置和启动硬盘操作等方面花了 1000 个时钟周期，并且在 DMA 传送完成后的中断处理需要 500 个时钟，则在硬盘 100%处于工作状态的情况下，CPU 花在硬盘 I/O 操作上的时间百分比大约是多少?

解：显然，从硬盘上读写 8KB 的数据，需要花费的时间为 $8KB/4MBps=2.048\times10^{-3}s\approx 2\times10^{-3}s$，所以如果硬盘一直处于工作状态的话，为了没有任何数据传输被错过，CPU 必须每秒钟大约有 $1/(2\times10^{-3})=0.5\times10^3$ 次 DMA 传送，因此，一秒钟内 CPU 用在硬盘I/O操作上的时钟周期数约为 $0.5\times10^3\times(1000+500)=750\times10^3$。因此，CPU 花费在硬盘 I/O 上的时间占整个 CPU 时间的百分比大约为 $750\times10^3/(500\times10^6)=1.5\times10^{-3}=0.15\%$。

采用 DMA 方式进行硬盘 I/O 时，在数据传送期间将不消耗任何处理器时钟周期，所以

即使硬盘一直在进行 I/O 操作，CPU 为它服务的时间也仅占 0.15%。事实上，硬盘在大多数时间内并不进行数据传送，而且每次 DMA 传输的数据块比 8KB 大得多，因此，CPU 为 I/O 所花费的时间会更少。当然，如果 CPU 同时要竞争存储器的话，存储器由于用于 DMA 传送，因而 CPU 会被延迟与存储器交换数据。但通过使用 cache，CPU 可避免大多数与 DMA 之间的访存冲突。因此，通常存储器带宽的大部分都可让给外设的 DMA 传送使用。

***4. DMA 与存储系统**

当 DMA 方式引入到 I/O 系统中时，存储系统和 CPU 之间的关系会变得更复杂。没有 DMA 控制器时，所有对存储器的访问都来自 CPU，通过 MMU 中的地址转换和 cache 访问来进行存储器存取。有了 DMA 后，系统中就有了另一个访问存储器的路径，它不通过地址转换机制和 cache 层次。这样，在虚拟存储器和 cache 系统中就会产生一些问题。这些问题的解决通常要结合硬件和软件两方面的技术支持。

因为在虚拟存储器系统中，页面数据同时有物理地址和虚拟地址，那么，DMA 是以虚拟地址还是以物理地址工作呢？很明显，若用虚拟地址，则在 DMA 接口中必须将页面的虚拟地址转换为物理地址；而使用物理地址所带来的问题是，每次 DMA 传送不能跨页。如果一个 I/O 请求跨页，那么，一次请求的一个数据块在送到主存时，就可能不在主存的一个连续的存储区中，因为每个虚页可以映射到主存的任意一个实页框中，所以多个连续的虚页不可能正好对应连续的实页框。因此，如果 DMA 采用物理地址的话，就必须限制所有的 DMA 传送都必须在一个页面之内进行。否则，DMA 就只能采用虚拟地址，在这种情况下，DMA 接口中应该有一个小的类似页表的地址映射表，用于将虚拟地址转换为物理地址。在 DMA 初始化的时候，由操作系统进行地址映射。这样的话，DMA 接口就不用关心传送数据在主存的具体位置了。另外一种方法是操作系统把一次传送分解成多次小数据量传送，每次只限定在一个物理页面内进行。

采用 cache 的系统中，一个数据项可能会产生两个副本，一个在 cache 中，一个在存储器中，因此具有 DMA 的 cache 系统也会产生问题。DMA 控制器直接向存储器发出访存请求而不通过 cache，这时，DMA 看到的一个主存单元的值与 CPU 看到的 cache 中的副本可能不同。考虑从磁盘中读一个数据，DMA 直接将其送到主存，如果有些被 DMA 写过的单元在 cache 中，那么，以后 CPU 读取这些单元的时候，就会得到一个老的值。类似地，如果 cache 采用写回(write-back)策略，当一个新的值在 cache 中写入时，这个值并未被马上写回存储器，而此时若 DMA 直接从主存读，那么读的可能是老的值。这个问题称为过时数据问题或 I/O 一致性问题。

解决 I/O 数据一致性问题的方法有三种。一种方式是让 I/O 活动通过 cache 进行，这样就保证了在 I/O 读时能读到最新的数据，而 I/O 写时能更新 cache 中的任何数据。当然，将所有 I/O 都通过 cache，其代价是非常大的。因为，一般 I/O 数据很少马上要用到，如果这样的数据把 CPU 正在使用的有用数据替换出去，那么就会影响 cache 的命中率，对 CPU 的性能带来很多负面影响。第二种方式是让操作系统在 I/O 读时有选择地使某些 cache 块无效，而在 I/O 写时迫使 cache 进行一次写回操作，这种操作经常被称为 cache 刷新，这种方式需要少量硬件支持。因为大部分 cache 刷新操作仅发生在 DMA 数据块访问时，而 DMA 访问又不经常发生，所以，如果软件能方便而有效地实现这种方法的话，可能是比较有效的一种。第三种方式是通过一个硬件机制来选择被刷新或使无效的 cache 项，这种用

硬件方式来保证 cache 一致性的方式大多被用在多处理器系统中。

*9.5.4 通道和 I/O 处理器方式

在大型计算机系统中，外围设备的数量、种类较多，为了在处理 I/O 请求时进一步减少中断处理次数和处理器占用时间，通常把对外设的管理和控制工作从 CPU 中分离出来，使 I/O 控制器更具智能化，这种 I/O 控制器称为通道控制器或 I/O 处理器。通道控制器和 I/O 处理器可独立执行一系列 I/O 操作，这些 I/O 操作序列被称为通道程序，这些程序可能被存储在通道或 I/O 处理器自己的存储器，或在共享的主存中，由通道或 I/O 处理器从主存中取出执行。当 CPU 执行到 I/O 请求时，操作系统要为 I/O 读写操作组织相应的传送参数或通道程序，通道或 I/O 处理器通过通道程序执行相应的操作，只有当整个通道程序都执行完后，才会中断 CPU。

1. 通道的基本概念

通道(Channel，简写为 CH)是一种专门的 I/O 控制器。通道方式与 DMA 方式的区别在于，DMA 方式通过 DMA 控制器控制总线，在外设和主存之间直接实现 I/O 传送；而通道通过执行通道程序对 I/O 操作管理。对 CPU 而言，通道比 DMA 具有更强的独立处理 I/O 的能力。DMA 控制器通常只控制一台或多台同类的高速设备；而通道可控制多台同类或不同类的设备。

在具有通道的系统中，通常采用主机—通道—设备控制器—外设四级结构，如图 9.50 所示。系统中可有多个通道，每个通道可接多个设备控制器，一个设备控制器可管理多个设备。

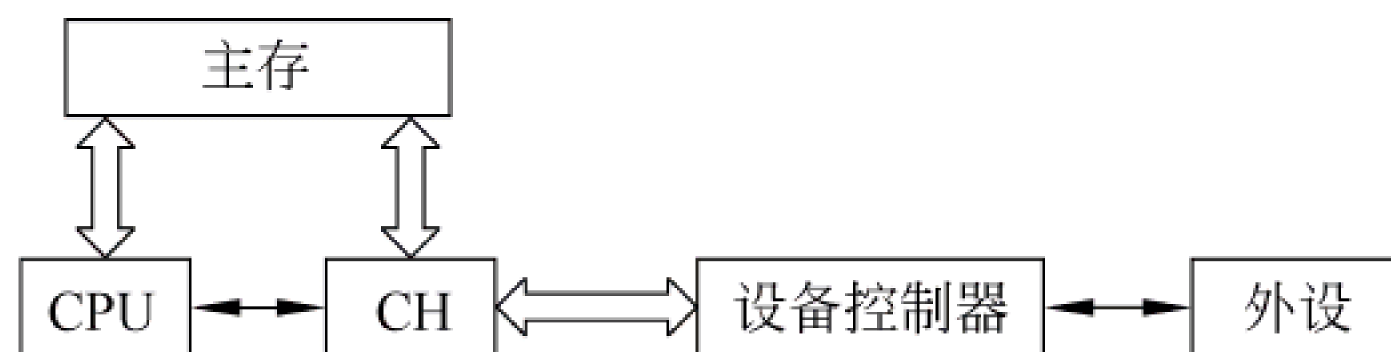


图 9.50 通道(CH)的位置

CPU 对通道的控制通过以下两种途径进行。

(1) 执行 I/O 指令

操作系统按约定的格式准备好命令和数据，编制好通道程序，当需要进行 I/O 操作时 CPU 通过执行 I/O 指令(例如 START I/O, TEST I/O，HALT I/O 等)来启动通道。通道被启动后，从主存指定单元开始处取出通道程序执行。I/O 指令是一种管态(特权)指令，在用户程序中不能使用。I/O 指令应给出通道开始工作所需的全部参数，如通道执行何种操作，在哪个通道和设备上进行操作等。I/O 指令和 CPU 的其他指令形式相同，由操作码和地址码组成，操作码表示执行何操作，地址码用来表示通道和设备的编号。CPU 启动通道后，通道和外部设备将独立进行工作，无须 CPU 干预。

(2) 处理来自通道的中断请求

当通道和外设发生异常或结束处理时，通道采用“中断”方式向 CPU 报告。通道是介于主机和设备控制器之间的机构，它一方面接受 CPU 的控制，向 CPU 发出相应的中断请求信号；另一方面它要负责对设备控制器的管理。

具体来说，它的基本职能如下：

① 接受 CPU 的 I/O 指令，按 CPU 的要求与指定的外设通信。

② 从内存读取通道程序并执行,从而向设备控制器和设备发送各种控制命令。

③ 组织外设和内存间的数据传送,并根据要求提供中间缓存。

④ 从外设得到设备的状态信息,并与通道本身的状态信息一起保存,并根据要求将状态信息送内存固定单元。

⑤ 将外设的中断请求和通道本身的中断请求,按序并及时报告给 CPU。

设备控制器类似于 I/O 接口,它接受通道的命令,并向设备发出控制信号,是通道对外设实现传输控制的执行机构。一个设备控制器可以控制多个同类的设备。

它的具体任务如下:

① 接受通道送来的通道命令,控制外设完成所要求的操作。

② 向通道反映外设的状态。

③ 将多种外设的不同信号转换为通道能识别的标准信号。

2. 通道的种类

按通道独立于 CPU 的程度来分,通道分为结合型通道和独立型通道。结合型通道在硬件上与 CPU 做在一起,而独立型通道则硬件上独立于 CPU。

通道按数据传送方式来分更有意义。通常将通道分为以下三种:字节多路通道、选择通道和成组多路通道。

(1) 字节多路通道

字节多路通道采用字节交叉传送方式进行数据传送,适合于连接多个低速 I/O 设备,这类设备传送一个字节的时间较短,而相邻字节之间数据准备时间较长。在字节多路通道的控制下,多台低速 I/O 设备以字节为单位,分时使用通道,轮流传送数据,实现多台 I/O 设备间的并行,以提高通道利用率,所以数据传输率是每个设备的传输率之和。字节多路通道由多个子通道构成,每个子通道并行工作,各服务于一个设备控制器。每个子通道中包含字符缓冲器、状态/控制寄存器以及通道参量(如字节计数值、内存地址等)、主存单元的地址指针等。

(2) 选择通道

选择通道用于对高速设备进行控制。对于高速设备,通道难以同时对多个设备进行控制,所以在一段时间内选择通道只执行一个设备的通道程序,为单个设备服务,采用“成组”方式传送。某个设备一旦被选中便独占通道,直到传送完毕才释放通道,所以传输速率高。选择通道可接多台同类设备,通道中包含一个参数寄存器,用于记录 I/O 操作所需的通道参量。

(3) 成组多路通道

通道在传输期间,只为一台高速设备服务,这是合理的,但有些高速设备机械辅助操作的时间较长,如磁盘寻道、磁带走带等,因此这期间的等待是一种浪费,可以考虑采用成组多路方式进行传送,多台设备以定长数据块为单位分时使用传输通路,轮流传送数据块。这种通道称为成组多路通道。它适用于控制磁盘机和磁带机之类的外设。

3. 通道程序

通道程序由若干通道命令字(CCW)构成,它是一组功能有限的 I/O 操作指令,能指定通道 I/O 操作所需的参量,并完成数据传送操作。CCW 一般包括操作命令码、数据在内存的首址、传送数据个数和控制标志字段等。

IBM 370 和 IBM 4300 的通道命令字如下所示。

0　　　　　7	8　　　　　31	32　　　　36	37　　39	40　　47	48　　　　63
操作码	内存地址	标志特征	000	未用	字计数值

上述通道命令字(CCW)是一个双字,主要由以下4个字段构成。

(1) 操作码字段:指出设备所进行的操作,一般包括以下几种操作命令。

① 读或反读操作:读操作时以地址递增顺序存储;反读操作时以递减顺序存储。

② 写操作:通道从主存取出数据,并将其写入外设。

③ 测试(取状态字)操作:外设送出它当前的状态信息和接口中产生的异常信息。

④ 转移(通道转移)操作:执行通道转移操作时,CCW从该命令给定的地址中取出,而不是按地址顺序从下一个单元中取出。因此,利用通道转移命令可以从一个通道命令序列转入另一个命令序列。

⑤ 控制操作:如磁盘寻道、磁带走带、卸带等。

(2) 内存地址字段:在执行读、反读、写、测试等操作时给出数据在主存中的首地址。

(3) 标志特征字段:共有以下5位标志特征位。

① 数据链特征CD和命令链特征CC:若CD=0、CC=0,则说明本指令是通道程序的最后一条指令。若CD=1、CC=0,则说明本指令与下一条指令的操作码相同。若CD=0、CC=1,则说明本指令和下条指令的操作码不同。

② 封锁错误长度特征SLI:为1时忽略长度错误。

③ 封锁写入主存特征SKIP:为1时只执行外设相应的操作,而不把数据写入主存。

④ 程序控制中断特征PCI:为1时产生I/O中断请求。

(4) 计数值字段:指定传送的数据个数。

下面给出一个例子来说明通道程序如何实现磁带与主存之间的数据传送。假定CPU调用时已由I/O指令启动了通道与磁带机,则通道从主存某固定单元中读出通道地址字CAW。然后再从CAW中取出通道程序首址,开始执行通道程序。表9.4是一个通道程序,共有5条通道命令字。

表9.4　通道程序举例

操作码	内存首址(十六进制)	标志特征位					字计数值(十进制)
		CD	CC	SLI	SKIP	PCI	
倒带		0	1	0	0	0	
走带	0003	0	1	0	0	0	
读	31B0	1	0	0	0	0	256
		1	0	0	1	0	512
读	5000	0	0	0	0	0	512

表9.4中的5条CCW的含义如下。

(1) 倒带:由于磁头可能处于磁带的中部,故应先倒带,使磁带反转到起始端。

(2) 走带:磁带正向越过3个数据块(记录区),但不读出。

(3) 读:读出256个字节的数据,写入首址为31B0H的主存缓冲区。

(4) 读带但不写入主存:由于第(3)条通道指令中,链接特征为 CD=1、CC=0,所以第(4)条指令的操作与第(3)条相同,仍为读出。但封锁写入主存位 SKIP=1,所以不写入主存。其作用相当于正向越过 256 字节的磁带长度。

(5) 读带:从磁带中读出 512 个字节的数据,写入首址为 5000H 的主存缓冲区。根据链接特征位,第 5 条是本通道程序的最后一条,至此结束。

4. 输入输出处理机

输入输出处理机方式是通道方式的进一步发展,有两种输入输出处理机系统结构。一种是通道结构的输入输出处理机,通常称为 I/O 处理机(IOP)。与通道方式一样,IOP 也通过执行通道程序对外设进行控制,能和 CPU 并行工作,提供了 DMA 控制能力。它与通道的区别主要如下。(1)通道只有功能有限的、面向外设控制和数据传送的指令系统;而 IOP 有自己专用的指令系统,不仅可用于进行外设控制和数据传送,而且可进行算术运算、逻辑运算、字节变换、测试等。(2)通道方式下,通道程序存于和 CPU 公用的主存中;而 IOP 有自己单独的存储器,并可访问系统内存。(3)通道方式下,许多工作仍然需要 CPU 实现;而 IOP 有自己的运算器和控制器,能处理传送出错及异常情况,能对传送的数据格式进行转换,能进行整个数据块的校验等。

具有 IOP 的 I/O 系统是一个分级的 I/O 系统。用户程序和 I/O 系统是隔离开的,用户程序只“看见”最高层。当 CPU 执行到用户程序中要求进行某种输入输出操作的指令时,它就发一个 I/O 请求,调用操作系统中的 I/O 管理程序。I/O 管理程序接受到 I/O 请求后,就组织这次 I/O 操作所需的数据/控制信息块,包括总线请求方式、总线物理宽度、通道操作命令字和通道程序的参数块(设备地址、数据地址、通道程序指针、回送结果单元等)等,并放在主存公共区域,然后启动 IOP 中的通道工作,IOP 从主存公共信息区读取 CPU 放在该处的控制信息,并根据 CPU 预先设置的信息选择一个指定的通道程序执行。

另一种输入输出处理机系统结构是外围处理器(PPU)方式,在大型计算机系统中,有时选用通用计算机担任 PPU,它基本上独立于主 CPU 而工作,也有自己的指令系统,可进行算术/逻辑运算、主存读写和与外设交换信息等。

输入输出处理机是一种特殊的装置,有特定的任务,所以相对于对等多处理器,其并行性很有限,它只用于传输信息、或对信息进行简单的加工和格式转换等。

至此,已经介绍了用于外设和主机之间进行数据传送的四种方法:程序查询、中断方式、DMA 控制以及通道和输入输出处理器方式。这些方法逐步把处理 I/O 操作的负担从 CPU 移到更智能化的 I/O 控制器或 I/O 处理器,使 CPU 时钟周期从 I/O 操作中释放出来。但也逐步增加了 I/O 系统的复杂性和价格,因此,一个给定的计算机系统应该选择它所连接的设备所合适的 I/O 控制方式。

9.6 本章小结

本章主要介绍输入输出系统的组成、外部设备的分类和特点、常用输入输出设备和外部存储设备的工作原理、I/O 接口的职能和分类、I/O 设备和主机的连接以及常用的几种输入输出方式,包括程序查询方式、程序中断方式和 DMA 方式等内容。

具体总结如下。

- 外部设备及其与主机的互连

  输入设备、输出设备和外存储器统称为外部设备，简称外设。所有外设通过相应的电缆连到I/O接口电路上，I/O接口电路再连到系统总线，最终与CPU和主存相连。

- 常用外设
  - 输入设备：键盘、鼠标。
  - 输出设备：打印机、显示器。
  - 外存储器：磁盘、磁带和光盘。
- 磁盘存储器的主要技术指标
  - 记录密度：道密度指单位长度上的磁道数；位密度为磁道中单位长度上的位数。
  - 平均存取时间：平均寻道时间和平均等待时间之和(数据传输时间相对较小，可忽略不计)。平均寻道时间指移动磁头到所读写磁道的平均时间；平均等待时间指要读写的扇区旋转到磁头下方所需的平均时间，等于磁盘旋转一圈所花时间的一半。
  - 数据传输率：分为内部数据传输率和外部数据传输率。内部数据传输率与磁盘转速有关，指寻道和旋转等待后，单位时间内从存储介质上读出或写入的二进制信息量。外部数据传输率与磁盘转速无关，指磁盘接口(磁盘控制器)从磁盘缓存读出或写入的数据传输率。
- 冗余磁盘阵列RAID
  - RAID技术的目的：增大容量，提高速度，并增强可靠性。
  - 小条带方式RAID的数据传输率高，但I/O响应速度慢，适合应用于流媒体播放系统等；大条区方式RAID则相反，适合应用于银行、证券等事务处理系统。
  - RAID级别有RAID 0～RAID 7，并派生出RAID 10、RAID 30和RAID 50等。目前广泛使用的是RAID 0、RAID 1和RAID 5，RAID 2未被使用，RAID 6也极少被使用。
- I/O接口的职能：数据缓冲、记录状态、传递命令、数据格式转换以及与主机侧和外设侧分别进行通信。
- I/O接口的类型：并行/串行、可编程/不可编程、通用/专用、轮询/中断/DMA、点对点/多点(总线式)。
- I/O端口：指I/O接口中的用户可访问寄存器，有数据端口、命令端口和状态端口。I/O端口的编址方式有两种。
  - 独立编址方式：对I/O端口单独编号，使它们成为一个独立的I/O地址空间。
  - 统一编址(存储器映射)方式：与主存地址空间统一编号，即将主存地址空间分出一部分地址给I/O端口进行编号。
- I/O指令：指CPU用来控制和访问I/O接口的操作指令。
- 常用I/O控制方式
  - 程序直接控制方式：分无条件传送和条件传送方式。
    - 无条件传送方式：利用程序定时传送数据，无须检测接口或设备的状态，适合于各类巡回检测或过程控制。

▲ 条件传送(程序查询)方式：CPU 通过查询外设接口中的“就绪(Ready)”、“忙(Busy)”和“完成(Done)”等状态来控制数据的传送，有定时查询和独占查询两种。

◆ 程序中断 I/O 方式：当外设准备好数据或准备好接收新数据或发生了特殊事件时，外设通过向 CPU 发中断请求来使 CPU 转到相应的中断服务程序去执行，在中断服务程序中完成数据交换或处理特殊事件。

◆ DMA 方式：适合像磁盘一类的高速设备(外存)以成批方式和主存直接交换数据。首先要对 DMA 控制器进行初始化；然后由 DMA 控制器控制总线在主存和高速设备之间进行直接数据交换；最后，DMA 控制器发出“DMA 传送结束”信号给外设接口，由外设接口发中断请求给 CPU，CPU 进行 DMA 结束处理。

习　题　9

1. 给出以下概念的解释说明。

(1) I/O 带宽	(2) 响应时间	(3) 编码键盘	(4) 非编码键盘
(5) 键盘扫描码	(6) 终端	(7) 磁道	(8) 柱面
(9) 扇区	(10) 道密度	(11) 位密度	(12) 平均存取时间
(13) 寻道时间	(14) 旋转等待(查找)时间	(15) 传输时间	(16) 数据传输率
(17) 磁盘控制器	(18) 冗余磁盘阵列(RAID)	(19) SCSI 总线	(20) I/O 接口
(21) I/O 控制器	(22) I/O 端口	(23) 命令(控制)端口	(24) 数据端口
(25) 状态端口	(26) I/O 空间	(27) 独立编址	(28) 统一编址
(29) 存储器映射 I/O	(30) 并行接口	(31) 串行接口	(32) 可编程接口
(33) 不可编程接口	(34) I/O 指令	(35) 程序查询 I/O	(36) 就绪状态
(37) 程序中断 I/O	(38) 可屏蔽中断	(39) 不可屏蔽中断	(40) 屏蔽字
(41) 中断响应优先级	(42) 中断处理优先级	(43) DMA 方式	(44) 周期挪用
(45) DMA 控制器	(46) 通道	(47) 通道指令	(48) 输入输出处理机

2. 简单回答下列问题。

(1) 什么是 I/O 接口？I/O 接口的基本功能有哪些？按数据传送方式分有哪两种接口类型？

(2) 串行接口和并行接口的特点各是什么？

(3) CPU 如何进行设备的寻址？I/O 端口的编址方式有哪两种？各有何特点？

(4) 什么是程序查询 I/O 方式？说明其工作原理。

(5) 什么是中断 I/O 方式？说明其工作原理。

(6) 什么叫向量中断？说明在向量中断方式下形成中断向量的基本方法。

(7) 对于向量中断，为什么 I/O 模块把中断请求设备标识放在总线的数据线上而不是放在地址线上？

(8) 在多周期处理器中并不是每个时钟周期后都允许响应中断。为什么？如果在一条指令执行过程中，CPU 为了响应中断而停止操作，会产生什么问题？

(9) 为什么在保护现场和恢复现场的过程中，CPU 必须关中断？

(10) DMA 方式能够提高成批数据交换效率的主要原因何在？

(11) DMA 方式和中断 I/O 方式有什么区别？试从请求对象、响应时机、适用设备等方面进行比较。

(12) 在 DMA 接口中，什么时候给出“DMA 请求”(或“总线请求”)信号？什么时候给出“中断请求”信号？CPU 在什么时候响应 DMA 请求？在什么时候响应中断请求？

3. 假定一个政府机构同时监控100路移动电话的通话消息,通话消息被分时复用到一个带宽为4MBps的网络上,复用使得每传送1KB的通话消息需额外开销150μs,若通话消息的采样频率为4kHz,每个样本的量化值占16位,要求计算每个通话消息的传输时间,并判断该网络带宽能否支持同时监控100路通话消息?

4. 假定一个程序重复完成将磁盘上一个4KB的数据块读出,进行相应处理后,写回到磁盘的另外一个数据区。各数据块内信息在磁盘上连续存放,并随机地位于磁盘的一个磁道上。磁盘转速为7200RPM,平均寻道时间为10ms,磁盘最大数据传输率为40MBps,磁盘控制器的开销为2ms,没有其他程序使用磁盘和处理器,并且磁盘读写操作和磁盘数据的处理时间不重叠。若程序对磁盘数据的处理需要20 000个时钟周期,处理器时钟频率为500MHz,则该程序完成一次数据块"读出-处理-写回"操作所需的时间为多少?每秒钟可以完成多少次这样的数据块操作?

5. 假定主存和磁盘存储器之间连接的同步总线具有以下特性:支持4字块和16字块两种长度(字长32位)的突发传送,总线时钟频率为200MHz,总线宽度为64位,每个64位数据的传送需一个时钟周期,向主存发送一个地址需要一个时钟周期,每个总线事务之间有两个空闲时钟周期。若访问主存时最初4个字的存取时间为200ns,随后每存取一个4字的时间是20ns,磁盘的数据传输率为5MBps,则在4字块和16字块两种传输方式下,该总线上分别最多可有多少个磁盘同时进行传输?

6. 假定有两个用来存储10TB数据的RAID系统。系统A使用RAID 1技术,系统B使用RAID 5技术。

(1) 系统A需要比系统B多用多少存储量?

(2) 假定一个应用需要向磁盘写入一块数据,若磁盘读或写一块数据的时间为30ms,则最坏情况下,在系统A和系统B上写入一块数据分别需要多长时间?

(3) 哪个系统更可靠?为什么?

7. 假定在一个使用RAID 5的系统中,采用先更新数据块、再更新校验块的信息更新方式。如果在更新数据块和更新校验块的操作之间发生了掉电现象,那么会出现什么问题?采用什么样的信息更新方式可避免这个问题?

8. 某终端通过RS-232串行通信接口与主机相连,采用起止式异步通信方式,若传输速率为1200波特,采用两相调制技术。通信协议为8位数据、无校验位、停止位为1位。则传送一个字节所需时间约为多少?若传输速度为2400波特,停止位为2位,其他不变,则传输一个字节的时间为多少?

9. 假定采用独立编址方式对I/O端口进行编号,那么,必须为处理器设计哪些指令来专门用于进行I/O端口的访问?连接处理器的总线必须提供哪些控制信号来表明访问的是I/O空间?

10. 假设有一个磁盘,每面有200个磁道,盘面总存储容量为1.6MB,磁盘旋转一周时间为25ms,每道有4个区,每两个区之间有一个间隙,磁头通过每个间隙需1.25ms。问:从该磁盘上读取数据时的最大数据传输率是多少(单位为字节/秒)?假如有人为该磁盘设计了一个与计算机之间的接口,如图9.51所示,磁盘每读出一位,串行送入一个移位寄存器,每当移满16位后向处理器发出一个请求取走数据的信号。在处理器响应该请求信号并读取移位寄存器内容的同时,磁盘继续读出一位一位数据并串行送入移位寄存器,如此继续工作。已知处理器在接到请求取走数据的信号以后,最长响应时间是3μs,这样设计的接口能否正确工作?若不能则应如何改进?

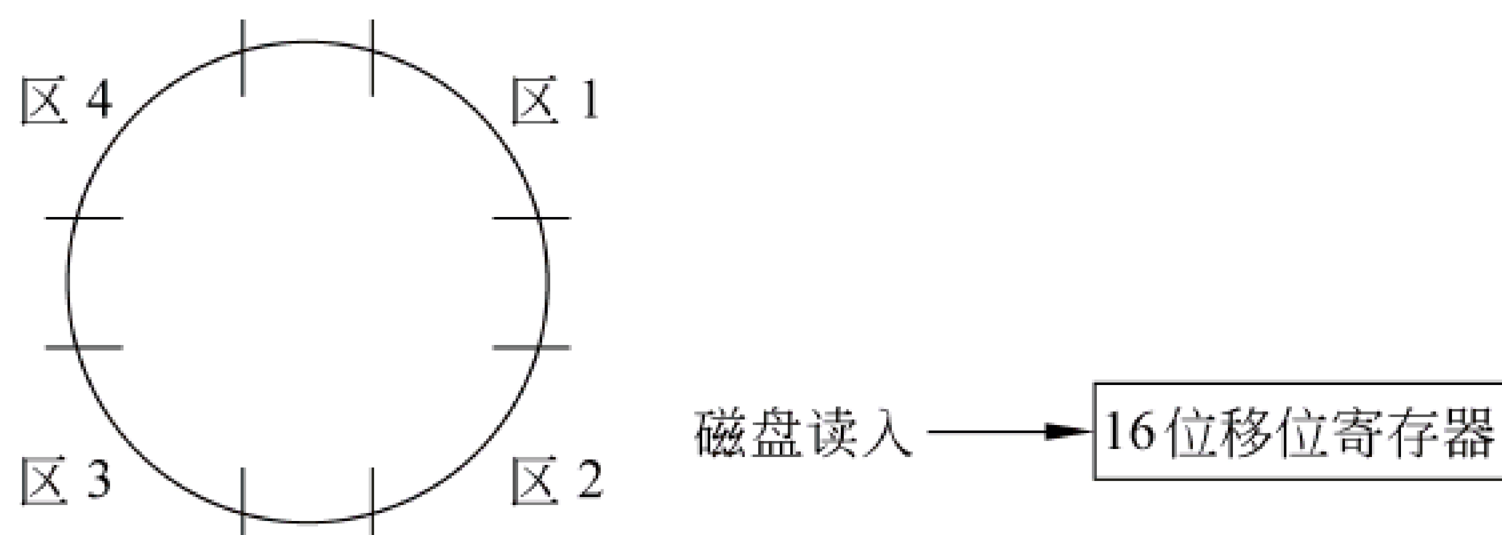


图9.51　题10用图

11. 假设某计算机带有 20 个终端同时工作，在运行用户程序的同时，能接受来自任意一个终端输入的字符信息，并将字符回送显示（或打印）。每一个终端的键盘输入部分有一个数据缓冲寄存器 RDBRi（i=1～20），当在键盘上按下某一个键时，相应的字符代码即进入 RDBRi，并使它的"完成"状态标志 Done$i$（i=1～20）置 1，要等处理器把该字符代码取走后，Done$i$ 标志才置 0。每个终端显示（或打印）输出部分也有一个数据缓冲寄存器 TDBRi（i=1～20），并有一个 Ready$i$（i=1～20）状态标志，该状态标志为 1 时，表示相应的 TDBRi 是空着的，准备接收新的输出字符代码，当 TDBRi 接收了一个字符代码后，Ready$i$ 标志才置 0，并将字符代码送到终端显示（或打印），为了接收终端的输入信息，处理器为每个终端设计了一个指针 PTRi（i=1～20）指向为该终端保留的主存输入缓冲区。处理器采用下列两种方案输入键盘代码，同时回送显示（或打印）。

(1) 每隔一固定时间 T 转入一个状态检查程序 DEVCHC，顺序地检查全部终端是否有任何键盘信息要输入，如果有，则顺序完成之。

(2) 允许任何有键盘信息输入的终端向处理器发出中断请求。全部终端采用共同的向量地址，利用它使处理器在响应中断后，转入一个中断服务程序 DEVINT，由后者询问各终端状态标志，并为最先遇到的请求中断的终端服务，服务结束后返回用户程序。

要求画出 DEVCHC 和 DEVINT 两个程序的流程图。

12. 假定某计算机的 CPU 主频为 500MHz，所连接的某个外设的最大数据传输率为 20kBps，该外设接口中有一个 16 位的数据缓存器，相应的中断服务程序的执行时间为 500 个时钟周期，则是否可以用中断方式进行该外设的输入输出？假定该外设的最大数据传输率改为 2MBps，则是否可以用中断方式进行该外设的输入输出？

13. 若某计算机有 5 级中断，中断响应优先级为 1＞2＞3＞4＞5，而中断处理优先级为 1＞4＞5＞2＞3。要求完成以下工作：

(1) 设计各级中断处理程序的中断屏蔽位（假设 1 为屏蔽，0 为开放）。

(2) 若在运行主程序时，同时出现第 2、4 级中断请求，而在处理第 2 级中断过程中，又同时出现 1、3、5 级中断请求，试画出此程序运行过程示意图。

14. 假定某计算机字长 16 位，没有 cache，运算器一次定点加法时间等于 100 毫微秒，配置的磁盘旋转速度为每分钟 3000 转，每个磁道上记录两个数据块，每一块有 8000 个字节，两个数据块之间间隙的越过时间为 2 毫秒，主存的存储周期为 500 毫微秒，存储器总线宽度为 16 位，总线带宽为 4MBps。请回答下列问题。

(1) 磁盘读写数据时的最大数据传输率是多少？

(2) 当磁盘按最大数据传输率与主机交换数据时，主存周期空闲百分比是多少？

(3) 直接寻址的"存储器-存储器"SS 型加法指令在无磁盘 I/O 操作打扰时的执行时间为多少？当磁盘 I/O 操作与一连串这种 SS 型加法指令执行同时进行时，则这种 SS 型加法指令的最快和最慢执行时间各是多少（假定采用多周期处理器方式，CPU 时钟周期等于主存周期）？

15. 假定某计算机所有指令都可用两个总线周期完成，一个总线周期用来取指令，另一个总线周期用来存取数据。总线周期为 250ns，因而，每条指令的执行时间为 500ns。若该计算机中配置的磁盘上每个磁道有 16 个 512 字节的扇区，磁盘旋转一圈的时间是 8.192ms，则采用周期挪用法进行 DMA 传送时，总线宽度为 8 位和 16 位的情况下该计算机指令执行速度分别降低了百分之几？

16. 假设一个主频为 1GHz 的处理器需要从某个成块传送的 I/O 设备读取 1000 字节的数据到主存缓冲区中，该 I/O 设备一旦启动即按 50kBps 的数据传输率向主机传送 1000 字节数据，每个字节的读取、处理并存入内存缓冲区需要 1000 个时钟周期，则以下 4 种方式下，在 1000 字节的读取过程中，CPU 用在该设备的 I/O 操作上的时间分别为多少？占整个处理器时间的百分比分别是多少？

(1) 采用定时查询方式，每次处理一个字节，一次状态查询至少需要 60 个时钟周期。

(2) 采用独占查询方式，每次处理一个字节，一次状态查询至少需要60个时钟周期。

(3) 采用中断I/O方式，外设每准备好一个字节发送一次中断请求。每次中断响应需要两个时钟周期，中断服务程序的执行需要1200个时钟周期。

(4) 采用周期挪用DMA方式，每挪用一次主存周期处理一个字节，一次DMA传送完成1000字节数据的I/O，DMA初始化和后处理的时间为2000个时钟周期，CPU和DMA没有访存冲突。

(5) 如果设备的速度提高到5MBps，则上述4种方式中，哪些是不可行的？为什么？对于可行的方式，计算出CPU花在该设备I/O操作上的时间占整个处理器时间的百分比？

(6) 如果外设不是成块传送设备，而是字符型设备，CPU每处理完一个字节后都要重新启动外设，外设在启动后0.02ms时准备好一个字节。每个字节的读取、处理(包括启动下次操作)并存入内存缓冲区还是1000个时钟周期，假定CPU总是在完成一个字节的读取后立即启动外设，则在(1)～(3)三种方式下，CPU花在该设备的I/O操作上的时间占整个处理器时间的百分比分别是多少？

参 考 文 献

[1] Randal E. Bryant，David R. O'Hallaron. Computer Systems：A Programmer's Perspective. Prentice Hall，2003

[2] David A. Patterson，John L. Hennessy. Computer Organization and Design：The Hardware/Software Interface，3rd Ed.. San Mateo，CA：Morgan Kaufman，2004

[3] John L. Hennessy，David A. Patterson. Computer Architecture：A Quantitative Approach，3rd Ed.. San Mateo，CA：Morgan Kaufman，2002

[4] Carl Hamacher，Zvonko Vranesic，Safwat Zaky. Computer Organization，5E.. The McGraw-Hill，2002

[5] William Stallings. Computer Organization and Architecture Design for Performance，7th Ed.. Prentice Hall，2006

[6] M. Morris Mano，Charies R. Kime. Logic and Computer Design Fundamentals，Third Edition. 北京：机械工业出版社，2008

[7] Randal E. Bryant，David R. O'Hallaron 著. 深入理解计算机系统（修订版）. 龚奕利，雷迎春译. 北京：中国电力出版社，2004

[8] David A. Patterson，John L. Hennessy 著. 计算机组成和设计 硬件/软件接口（第 3 版）. 郑纬民等译. 北京：机械工业出版社，2007

[9] Carl Hamacher，Zvonko Vranesic，Safwat Zaky 著. 计算机组织（第 5 版）. 张红光，张健民，李莹等译. 北京：机械工业出版社，2004

[10] Yale N. Patt，Sanjay J. Patel 著. 计算机系统概论（第 2 版）. 梁阿磊，蒋兴昌，林凌译. 北京：机械工业出版社，2007

[11] 王诚，刘卫东，宋佳兴. 计算机组成与设计（第 3 版）. 北京：清华大学出版社，2008

[12] 杨厚俊，张公敬，张昆藏. 计算机系统结构—奔腾 PC（第二版）. 北京：科学出版社，2004

[13] Dominic Sweetman 著. MIPS 体系结构透视. 李鹏，鲍峥，石洋等译. 北京：机械工业出版社，2008

[14] Andrew S. Tanenbaum 著. 计算机组成结构化方法（第 5 版）. 刘卫东，宋佳兴译. 北京：人民邮电出版社，2006

[15] 薛宏熙，胡秀珠. 计算机组成与设计. 北京：清华大学出版社，2007

[16] 白中英，戴志涛，杨春武，张乐天，于艳丽. 计算机组织与体系结构. 北京：清华大学出版社，2008

[17] 蒋本珊. 计算机组成原理教师用书. 北京：清华大学出版社，2005

[18] 张功萱，顾一禾，邹建伟，王晓峰. 计算机组成原理. 北京：清华大学出版社，2005

[19] 美国 UC Berkeley 大学"Machine Structure"08 年课程网站：http://inst. eecs. berkeley. edu/～cs61c/su08/

[20] 美国 UC Berkeley 大学"Components and Design Techniques for Digital System"09 年课程网站：http://inst. eecs. berkeley. edu/～cs150/sp09/

[21] 美国 UC Berkeley 大学"Computer Architecture and Engineering"09 年课程网站：http://inst. eecs. berkeley. edu. /～cs152/sp09/

[22] 美国 Stanford 大学"Computer Organization and Systems"09 年课程网站：Http://www. stanford. edu/calss/cs107/

[23] 美国 Stanford 大学“Digital Systems II”09 年课程网站：http://www.stanford.edu/class/ee108b/
[24] 美国 Carnegie Mellon 大学“Introduction to Computer Architecture”09 年课程网站：http://www.ece.cmu.edu/～ece447/
[25] 美国 Univ. Illinois at Urbana-Champaign“Computer Architecture Ⅱ”08 年课程网站：http://www.cs.uiuc.edu/class/sp08/cs232/
[26] 美国麻省理工学院(MIT)“Computation Structures”09 年课程网站：http://6004.csail.mit.edu



普通高等教育“十一五”国家级规划教材
21世纪大学本科计算机专业系列教材

近期出版书目

- 计算机导论(第2版)
- 程序设计导引及在线实践
- 程序设计基础
- 程序设计基础习题解析与实验指导
- 离散数学(第2版)
- 离散数学习题解答与学习指导(第2版)
- 数据结构与算法
- 形式语言与自动机理论(第2版)
- 形式语言与自动机理论教学参考书(第2版)
- 计算机组成原理(第2版)
- 计算机组成原理教师用书(第2版)
- 计算机组成原理学习指导与习题解析(第2版)
- 计算机组成与体系结构
- 微型计算机系统与接口
- 计算机操作系统
- 计算机操作系统学习指导与习题解答
- 计算机组成与系统结构
- 数据库系统原理
- 编译原理
- 软件工程
- 计算机图形学
- 计算机网络(第2版)
- 计算机网络教师用书(第2版)
- 计算机网络实验指导书(第2版)
- 计算机网络习题集与习题解析(第2版)
- 计算机网络软件编程指导书
- 人工智能
- 多媒体技术原理及应用(第2版)
- 算法设计与分析(第2版)
- 算法设计与分析习题解答(第2版)
- C++程序设计
- 面向对象程序设计
- 计算机网络工程
- 计算机网络工程实验教程
- 信息安全原理及应用

读者意见反馈

亲爱的读者：

感谢您一直以来对清华版计算机教材的支持和爱护。为了今后为您提供更优秀的教材，请您抽出宝贵的时间来填写下面的意见反馈表，以便我们更好地对本教材做进一步改进。同时如果您在使用本教材的过程中遇到了什么问题，或者有什么好的建议，也请您来信告诉我们。

地址：北京市海淀区双清路学研大厦 A 座 602 室 计算机与信息分社营销室 收

邮编：100084　　　　电子邮件：jsjjc@tup.tsinghua.edu.cn

电话：010-62770175-4608/4409　　　　邮购电话：010-62786544

教材名称：计算机组成与系统结构

ISBN：978-7-302-21905-7

个人资料

姓名：________ 年龄：______ 所在院校/专业：____________

文化程度：________ 通信地址：____________________

联系电话：________ 电子信箱：____________________

您使用本书是作为： □指定教材 □选用教材 □辅导教材 □自学教材

您对本书封面设计的满意度：

□很满意 □满意 □一般 □不满意 改进建议____________

您对本书印刷质量的满意度：

□很满意 □满意 □一般 □不满意 改进建议____________

您对本书的总体满意度：

从语言质量角度看 □很满意 □满意 □一般 □不满意

从科技含量角度看 □很满意 □满意 □一般 □不满意

本书最令您满意的是：

□指导明确 □内容充实 □讲解详尽 □实例丰富

您认为本书在哪些地方应进行修改？（可附页）

____

____

您希望本书在哪些方面进行改进？（可附页）

____

____

电子教案支持

敬爱的教师：

为了配合本课程的教学需要，本教材配有配套的电子教案（素材），有需求的教师可以与我们联系，我们将向使用本教材进行教学的教师免费赠送电子教案（素材），希望有助于教学活动的开展。相关信息请拨打电话 010-62776969 或发送电子邮件至 jsjjc@tup.tsinghua.edu.cn 咨询，也可以到清华大学出版社主页（http://www.tup.com.cn 或 http://www.tup.tsinghua.edu.cn）上查询。